Today’s Plan

Recap

Searching algorithms and their analysis

Sorting algorithms and their analysis
Announcements

Questions?
Searching

Looking for something!
In this discussion we will assume searching for an element in an array
Linear search

Most intuitive
Start at first position and keep looking until you find it

```c
int linearSearch(int a[], int size, int value) {
    for (int i = 0; i < size; i++) {
        if (a[i] == value) {
            return i;
        }
    }
    return -1;
}
```
How long does linear search take?

If you assume value is in the array and probability of finding it at any location is uniform, on average $\frac{n}{2}$.

If value is not in the array (worst case) $n$.

Either way, it’s $O(n)$. 
What if you know *array is sorted*?
Can you do better than linear search?
You are given a sorted array of integers.

How would you search for 115? (try to do it in fewer than n steps: don’t search sequentially)

You can write pseudocode or succinctly explain your algorithm
We have done this before!
When?
Look in ?
Binary Search
Binary Search

3 14 43 76 100 108 158 195 200 274 523 543 599
Binary Search
Binary Search
Binary Search

3  14  43  76  100  108  158  195  200  274  523  543  599
Binary Search
Binary Search

3 14 43 76 100 108 158 195 200 274 523 543 599
Binary Search

What is happening here?
Binary Search

What is happening here?

Size of search is cut in half at each step
Binary Search

What is happening here?

Size of search is cut in half at each step

Let $T(n)$ be the running time and assume $n = 2^k$

$T(n) = T(n/2) + 1$

The running time

Simplification: assume $n$ is a power of 2 so it can be evenly divided in two parts

One comparison

Search lower OR upper half
Binary Search

What is happening here?

Size of search is cut in half at each step

Let \( T(n) \) be the running time and assume \( n = 2^k \)

\[
T(n) = T(n/2) + 1
\]

\[
T(n/2) = T(n/4) + 1
\]

One comparison

Search lower OR upper half of \( n/2 \)
Binary Search

What is happening here?

Size of search is cut in half at each step

Let $T(n)$ be the running time and assume $n = 2^k$

\[ T(n) = T(n/2) + 1 \]
\[ T(n/2) = T(n/4) + 1 \]
\[ T(n) = T(n/4) + 1 + 1 \]
Binary Search

What is happening here?

Size of search is cut in half at each step

Let $T(n)$ be the running time and assume $n = 2^k$

$T(n) = T(n/2) + 1$

$T(n) = T(n/4) + 2$

$\ldots$

$T(n) = T(n/2^k) + k$
Binary Search

What is happening here?

Size of search is **cut in half** at each step

Let $T(n)$ be the running time and **assume** $n = 2^k$

$T(n) = T(n/2) + 1$

$T(n) = T(n/4) + 2$

$\ldots$

$T(n) = T(n/2^k) + k$
Binary Search

What is happening here?

Size of search is \textbf{cut in half} at each step

Let $T(n)$ be the running time and assume $n = 2^k$

$T(n) = T(n/2) + 1$

$T(n) = T(n/4) + 2$

\ldots

$T(n) = T(n/2^k) + k$

$T(n) = T(1) + \log_2(n)$

$n/n = 1$

The number to which I need to raise 2 to get $n$
And we said $n = 2^k$
Binary Search

What is happening here?

Size of search is **cut in half** at each step

Let $T(n)$ be the running time and assume $n = 2^k$

$T(n) = T(n/2) + 1$

$T(n) = T(n/4) + 2$

\[ \ldots \]

$T(n) = T(n/2^k) + k$

$T(n) = T(1) + \log_2(n)$

Binary search is $O(\log(n))$
Sorting

Rearranging a sequence into increasing (decreasing) order!
Several approaches

Can do it in many ways

What is the best way?

Let’s find out using Big-O
Lecture Activity

Write **pseudocode** to sort an array.
There are many approaches to sorting. We will look at some comparison-based approaches here.
Selection Sort
Selection Sort

Find smallest element and move it at lowest position.

1st Pass
Selection Sort

Find smallest element and move it at lowest position

1st Pass

Swap
Selection Sort

Find smallest element and move it at lowest position

1st Pass
Selection Sort

Find smallest element and move it at lowest position

Unsorted

Sorted

2nd Pass
Selection Sort

Find smallest element and move it at lowest position

Swap

2nd Pass
Selection Sort

Find smallest element and move it at lowest position

Unsorted

Sorted

2nd Pass
Find smallest element and move it at lowest position
Selection Sort

Find smallest element and move it at lowest position

3rd Pass
Selection Sort

Find smallest element and move it at lowest position

Unsorted

Sorted

4th Pass
Selection Sort

Find smallest element and move it at lowest position

4th Pass
Selection Sort

Find smallest element and move it at lowest position

Unsorted

Sorted

5th Pass
Selection Sort

Find smallest element and move it at lowest position

5th Pass

Swap

Unsorted

Sorted
Selection Sort

Find smallest element and move it at lowest position

5th Pass
Selection Sort

Find smallest element and move it at lowest position

6th Pass
Selection Sort

Find smallest element and move it at lowest position
Selection Sort

Find the smallest item and move it at position 1

Find the next-smallest item and move it at position 2

...
Selection Sort Analysis

How much work?

Find smallest: look at \( n \) elements
Selection Sort Analysis

How much work?

Find smallest: look at $n$ elements

Find second smallest: look at $n-1$ elements
Selection Sort Analysis

How much work?

Find smallest: look at \( n \) elements

Find second smallest: look at \( n-1 \) elements

Find third smallest: look at \( n-2 \) elements

...
Selection Sort Analysis

How much work?

Find smallest: look at $n$ elements

Find second smallest: look at $n-1$ elements

Find third smallest: look at $n-2$ elements

. . .

Total work: $n + (n-1) + (n-2) + \ldots + 1$
\[ n + (n-1) + \ldots + 2 + 1 = \frac{n(n+1)}{2} \]
Selection Sort Analysis

$T(n) = n(n+1) / 2$ comparisons + $n$ data moves = $O(\ )$?
Selection Sort Analysis

\[ T(n) = \frac{n(n+1)}{2} \text{ comparisons} + n \text{ data moves} = \mathcal{O}(n^2) \]

\[ T(n) = \frac{(n^2+n)}{2} + n = \mathcal{O}(n^2) \]
Selection Sort Analysis

\[ T(n) = \frac{n(n+1)}{2} \text{ comparisons} + n \text{ data moves} = O(\text{?}) \]

\[ T(n) = \frac{n^2+n}{2} + n = O(\text{?}) \]

Ignore constant

Ignore non-dominant terms
Selection Sort Analysis

\[ T(n) = n(n+1)/2 \text{ comparisons} + n \text{ data moves} = \Omega()? \]

\[ T(n) = (n^2+n)/2 + n = \Omega(n^2) \]

Ignore constant

Ignore non-dominant terms
Selection Sort Analysis

\[ T(n) = \frac{n(n+1)}{2} \text{ comparisons} + n \text{ data moves} = O()? \]

\[ T(n) = \frac{(n^2+n)}{2} + n = O(n^2) \]

Selection Sort run time is \( O(n^2) \)
template<
class T>
void selectionSort(T the_array[], int n)
{
    // last = index of the last item in the subarray of items yet
    // to be sorted;
    // largest = index of the largest item found
    for (int last = n - 1; last >= 1; last--)
    {
        // At this point, the_array[last+1..n-1] is sorted, and its
        // entries are greater than those in the_array[0..last].
        // Select the largest entry in the_array[0..last]
        int largest = findIndexOfLargest(the_array, last+1);

        // Swap the largest entry, the_array[largest], with
        // the_array[last]
        std::swap(the_array[largest], the_array[last]);
    } // end for
} // end selectionSort
template<class T>
void selectionSort(T the_array[], int n)
{
    // last = index of the last item in the subarray of items yet
    //       to be sorted;
    // largest = index of the largest item found
    for (int last = n - 1; last >= 1; last--)
    {
        // At this point, the_array[last+1..n-1] is sorted, and its
        // entries are greater than those in the_array[0..last].
        // Select the largest entry in the_array[0..last]
        int largest = findIndexOfLargest(the_array, last+1);

        // Swap the largest entry, the_array[largest], with
        // the_array[last]
        std::swap(the_array[largest], the_array[last]);
    } // end for
} // end selectionSort

O( n^2)
Stability

A sorting algorithm is **Stable** if elements that are equal remain in the same order relative to each other after sorting.
Selection Sort

Find smallest element and move it at lowest position
Selection Sort

Find smallest element and move it at lowest position
Selection Sort

Find smallest element and move it at lowest position

Swap
Selection Sort

Find smallest element and move it at lowest position
Selection Sort Analysis

Execution time DOES NOT depend on initial arrangement of data => \textbf{ALWAYS} $O(n^2)$

$O(n^2)$ comparisons

Good choice for \textit{small $n$ and/or data moves are costly} (\textit{O(n) data moves})

Unstable
Understanding $O(n^2)$
Understanding $O(n^2)$

$T(n) \approx 4T(n)$

$(2n)^2 = 4n^2$
Understanding $O(n^2)$

$T(n)$

$T(3n) \approx 9T(n)$

$(3n)^2 = 9n^2$
Understanding $O(n^2)$ on large input

If size of input increases by factor of 100
Execution time increases by factor of 10,000
$T(100n) = 10,000T(n)$
Understanding $O(n^2)$ on large input

If size of input increases by factor of 100
Execution time increases by factor of 10,000
$T(100n) = 10,000T(n)$

Assume $n = 100,000$ and $T(n) = 17$ seconds
Sorting $10,000,000$ takes $10,000$ longer
Understanding $O(n^2)$ on large input

If size of input increases by factor of 100
Execution time increases by factor of 10,000
$T(100n) = 10,000T(n)$

Assume $n = 100,000$ and $T(n) = 17$ seconds
Sorting 10,000,000 takes 10,000 longer

Sorting 10,000,000 entries takes $\approx 2$ days

Multiplying input by 100 to go from 17 sec to 2 days!!!
Raise your hand if you had Selection Sort
Bubble Sort
Bubble Sort

Compare adjacent elements and if necessary swap them.
Bubble Sort

Compare adjacent elements and if necessary swap them

1st Pass

Unsorted

Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them.

Swap

1st Pass

Unsorted

Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them

1st Pass
Bubble Sort

Compare adjacent elements and if necessary swap them

Swap

1st Pass

Unsorted

Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them

1st Pass

Unsorted

Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them

Swap

1st Pass

Unsorted

Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them

1st Pass
Compare adjacent elements and if necessary swap them.

Bubble Sort

1st Pass

Unsorted

Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them.

1st Pass

Swap
Bubble Sort

End of 1st Pass:
Not sorted, but largest has "bubbled up" to its proper position

Compare adjacent elements and if necessary swap them
Bubble Sort

Compare adjacent elements and if necessary swap them

2nd Pass:
Sort n-1
Bubble Sort

Compare adjacent elements and if necessary swap them

2nd Pass

Unsorted

Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them.
Bubble Sort

Compare adjacent elements and if necessary swap them

Swap

2nd Pass

Unsorted

Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them

2nd Pass

Unsorted
Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them.
Bubble Sort

Compare adjacent elements and if necessary swap them

3rd Pass: Sort n-2
Bubble Sort

Compare adjacent elements and if necessary swap them.
Bubble Sort

Compare adjacent elements and if necessary swap them

Array is sorted
But our algorithm doesn’t know
It keeps on going
Bubble Sort

Compare adjacent elements and if necessary swap them

3rd Pass

Unsorted

Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them

3rd Pass

Unsorted

Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them

4th Pass:
Sort n-3
Bubble Sort

Compare adjacent elements and if necessary swap them

4th Pass

Unsorted
Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them

4th Pass

Unsorted

Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them

5th Pass:
Sort n-4
Bubble Sort

Compare adjacent elements and if necessary swap them

5th Pass

Unsorted

Sorted
Bubble Sort

Compare adjacent elements and if necessary swap them

Done!
Bubble Sort Analysis

How much work?

First pass: \textbf{n-1} comparisons and \textbf{at most n-1} swaps

Second pass: \textbf{n-2} comparisons and \textbf{at most n-2} swaps

Third pass: \textbf{n-3} comparisons and \textbf{at most n-3} swaps

\ldots

Total work: \((n-1) + (n-2) + \ldots + 1\)
\[(n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}\]
Bubble Sort Analysis

\[ T(n) = \frac{n(n-1)}{2} \text{ comparisons} + \frac{n(n-1)}{2} \text{ swaps} = O(\ )? \]

A swap is usually more than one operation but this simplification does not change the analysis

\[ T(n) = 2\left(\frac{n(n-1)}{2}\right) = O(\ )? \]
Bubble Sort Analysis

\[ T(n) = \frac{n(n-1)}{2} \text{ comparisons} + \frac{n(n-1)}{2} \text{ swaps} = O(\ )? \]

A swap is usually more than one operation but this simplification does not change the analysis

\[ T(n) = 2\left( \frac{n(n-1)}{2} \right) = O(\ )? \]

\[ T(n) = 2\left( \frac{n^2-n}{2} \right) = O(\ )? \]
Bubble Sort Analysis

\[ T(n) = \frac{n(n-1)}{2} \text{ comparisons} + \frac{n(n-1)}{2} \text{ swaps} = \Theta()? \]

A swap is usually more than one operation but this simplification does not change the analysis

\[ T(n) = 2 \left( \frac{n(n-1)}{2} \right) = \Theta()? \]

\[ T(n) = 2 \left( \frac{n^2-n}{2} \right) = \Theta()? \]

\[ T(n) = n^2-n = \Theta()? \]

Ignore non-dominant terms
Bubble Sort Analysis

\[ T(n) = \frac{n(n-1)}{2} \text{ comparisons} + \frac{n(n-1)}{2} \text{ swaps} = O(\cdot)? \]

A swap is usually more than one operation but this simplification does not change the analysis.

\[ T(n) = 2\left( \frac{n(n-1)}{2} \right) = O(\cdot)? \]

\[ T(n) = 2\left( \frac{n^2-n}{2} \right) = O(\cdot)? \]

\[ T(n) = n^2-n = O( n^2) \]

Bubble Sort run time is \( O(n^2) \)
Optimize!

Easy to check:
if there are no swaps in any given pass
stop because it is sorted
Bubble Sort

Compare adjacent elements and if necessary swap them
Bubble Sort

Compare adjacent elements and if necessary swap them.
Bubble Sort

Compare adjacent elements and if necessary swap them.
Bubble Sort

Compare adjacent elements and if necessary swap them
Bubble Sort

Compare adjacent elements and if necessary swap them
Bubble Sort

Compare adjacent elements and if necessary swap them
Bubble Sort

Compare adjacent elements and if necessary swap them
template<class T>
void bubbleSort(T the_array[], int n)
{
    bool sorted = false; // False when swaps occur
    int pass = 1;
    while (!sorted && (pass < n))
    {
        // At this point, the_array[n+1-pass..n-1] is sorted
        // and all of its entries are > the entries in the_array[0..n-pass]
        sorted = true; // Assume sorted
        for (int index = 0; index < n - pass; index++)
        {
            // At this point, all entries in the_array[0..index-1]
            // are <= the_array[index]
            int nextIndex = index + 1;
            if (the_array[index] > the_array[nextIndex])
            {
                // Exchange entries
                std::swap(the_array[index], the_array[nextIndex]);
                sorted = false; // Signal exchange
            }
            // end if
        }
        // end for
        // Assertion: the_array[0..n-pass-1] < the_array[n-pass]
        pass++;
    }
    // end while
} // end bubbleSort
template<class T>
void bubbleSort(T the_array[], int n)
{
    bool sorted = false; // False when swaps occur
    int pass = 1;
    while (!sorted && (pass < n))
    {
        // At this point, the_array[n+1-pass..n-1] is sorted
        // and all of its entries are > the entries in the_array[0..n-pass]
        sorted = true; // Assume sorted
        for (int index = 0; index < n - pass; index++)
        {
            // At this point, all entries in the_array[0..index-1]
            // are <= the_array[index]
            int nextIndex = index + 1;
            if (the_array[index] > the_array[nextIndex])
            {
                // Exchange entries
                std::swap(the_array[index], the_array[nextIndex]);
                sorted = false; // Signal exchange
            } // end if
        } // end for
        // Assertion: the_array[0..n-pass-1] < the_array[n-pass]
        pass++;
    } // end while
} // end bubbleSort
Bubble Sort Analysis

Execution time DOES depend on initial arrangement of data

**Worst case:** $O(n^2)$ comparisons and data moves

**Best case:** $O(n)$ comparisons and data moves

Stable

If array is already sorted bubble sort will stop after first pass and no swaps => good choice for small $n$ and data likely somewhat sorted
Raise your hand if you had Bubble Sort
Insertion Sort
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

1st Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

1st Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region.
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region.
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

2nd Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

Swap

2nd Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

2nd Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Pick first element in unsorted region and put it in right place in sorted region.
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

Unsorted

Sorted

3rd Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

Swap

3rd Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

3rd Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

Swap

Unsorted

Sorted

3rd Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

3rd Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

Unsorted

Sorted

4th Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

4th Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

5th Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

5th Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

5th Pass
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region.
Insertion Sort Analysis

How much work?

First pass: 1 comparison and at most 1 swap

Second pass: at most 2 comparisons and at most 2 swaps

Third pass: at most 3 comparisons and at most 3 swaps

. . .

Total work: $1 + 2 + 3 + \ldots + (n-1)$
\[ 1 + 2 + \ldots + (n-2) + (n-1) = \frac{n(n-1)}{2} \]
Insertion Sort Analysis

\[ T(n) = n(n-1)/2 \text{ comparisons } + n(n-1)/2 \text{ swaps} = O(\ )? \]

\[ T(n) = 2( (n^2-n)/2 ) = O(\ )? \]

\[ T(n) = n^2-n = O( n^2) \]

Insertion Sort run time is \( O(n^2) \)
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region.
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region.
Pick first element in unsorted region and put it in right place in sorted region.
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region

Unsorted
Sorted
Insertion Sort

Pick first element in unsorted region and put it in right place in sorted region
Insertion Sort Analysis

Execution time DOES depend on initial arrangement of data

**Worst case:** $O(n^2)$ comparisons and data moves

**Best case:** $O(n)$ comparisons and data moves

**Stable**

If array is already sorted Insertion sort will do only $n$ comparisons and no swaps => good choice for small $n$ and data likely somewhat sorted
template<class T>
void insertionSort(T the_array[], int n)
{
    // unsorted = first index of the unsorted region,
    // loc = index of insertion in the sorted region,
    // next_item = next item in the unsorted region.
    // Initially, sorted region is the_array[0],
    // unsorted region is the_array[1..n-1].
    // In general, sorted region is the_array[0..unsorted-1],
    // unsorted region the_array[unsorted..n-1]
    for (int unsorted = 1; unsorted < n; unsorted++)
    {
        // At this point, the_array[0..unsorted-1] is sorted.
        // Find the right position (loc) in the_array[0..unsorted]
        // for the_array[unsorted], which is the first entry in the
        // unsorted region; shift, if necessary, to make room
        T next_item = the_array[unsorted];
        int loc = unsorted;
        while ((loc > 0) && (the_array[loc - 1] > next_item))
        {
            // Shift the_array[loc - 1] to the right
            the_array[loc] = the_array[loc - 1];
            loc--;
        } // end while

        // At this point, the_array[loc] is where next_item belongs
        the_array[loc] = next_item; // Insert next_item into sorted region
    } // end for
} // end insertionSort
template<class T>
void insertionSort(T the_array[], int n)
{
    // unsorted = first index of the unsorted region,
    // loc = index of insertion in the sorted region,
    // next_item = next item in the unsorted region.
    // Initially, sorted region is the_array[0],
    // unsorted region is the_array[1..n-1].
    // In general, sorted region is the_array[0..unsorted-1],
    // unsorted region the_array[unsorted..n-1]
    for (int unsorted = 1; unsorted < n; unsorted++)
    {
        // At this point, the_array[0..unsorted-1] is sorted.
        // Find the right position (loc) in the_array[0..unsorted]
        // for the_array[unsorted], which is the first entry in the
        // unsorted region; shift, if necessary, to make room
        T next_item = the_array[unsorted];
        int loc = unsorted;
        O(n)
        while ((loc > 0) && (the_array[loc - 1] > next_item))
        {
            // Shift the_array[loc - 1] to the right
            the_array[loc] = the_array[loc - 1];
            loc--;
        } // end while

        // At this point, the_array[loc] is where next_item belongs
        the_array[loc] = next_item; // Insert next_item into sorted region
    } // end for
} // end insertionSort

O( n^2)
Raise your hand if you had Insertion Sort
What we have so far

<table>
<thead>
<tr>
<th></th>
<th>Worst Case</th>
<th>Best Case</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Selection Sort</strong></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td><strong>Bubble Sort</strong></td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>Insertion Sort</strong></td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Lecture Activity

Sort the array using **Insertion Sort**
Show the entire array after each comparison/swap operation and at each step mark clearly the division between the sorted and unsorted portions of the array.

Pick first element in unsorted region and put it in right place in sorted region.

- Initial array: 5, 8, 3, 4, 9, 2, 7
- After first comparison: 5, 8, 3, 4, 9, 2, 7
- After second comparison: 5, 8, 3, 4, 9, 2, 7
- After third comparison: 5, 8, 3, 4, 9, 2, 7
- After fourth comparison: 5, 8, 3, 4, 9, 2, 7
- After fifth comparison: 5, 8, 3, 4, 9, 2, 7
- After sixth comparison: 5, 8, 3, 4, 9, 2, 7
- After seventh comparison: 5, 8, 3, 4, 9, 2, 7
- After eighth comparison: 5, 8, 3, 4, 9, 2, 7
- After ninth comparison: 5, 8, 3, 4, 9, 2, 7
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
https://www.toptal.com/developers/sorting-algorithms

<table>
<thead>
<tr>
<th>Play All</th>
<th>Insertion</th>
<th>Selection</th>
<th>Bubble</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nearly Sorted</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reversed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Can we do better?
Can we do better?

**Divide and Conquer!!!**

![Diagram showing the divide and conquer strategy with subproblems and computations](image)
Merge Sort
Understanding $O(n^2)$

| 100 | 14 | 3 | 43 | 200 | 274 | 523 | 108 | 76 | 195 | 599 | 158 | 2 | 260 | 11 | 64 | 932 | 5 |

$T(n)$
Understanding $O(n^2)$

| 100 | 14 | 3 | 43 | 200 | 274 | 523 | 108 | 76 | 195 | 599 | 158 | 2 | 260 | 11 | 64 | 932 | 5 |

$T(n)$

| 100 | 14 | 3 | 43 | 200 | 274 | 523 | 108 | 76 |

| 195 | 599 | 158 | 2 | 260 | 11 | 64 | 932 | 5 |
Understanding $O(n^2)$

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>14</td>
<td>3</td>
<td>43</td>
<td>200</td>
<td>274</td>
<td>523</td>
<td>108</td>
<td>76</td>
<td>195</td>
<td>599</td>
<td>158</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>14</td>
<td>3</td>
<td>43</td>
<td>200</td>
<td>274</td>
<td>523</td>
<td>108</td>
<td>76</td>
<td>195</td>
<td>599</td>
<td>158</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>14</td>
<td>3</td>
<td>43</td>
<td>200</td>
<td>274</td>
<td>523</td>
<td>108</td>
<td>76</td>
<td>195</td>
<td>599</td>
<td>158</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>14</td>
<td>3</td>
<td>43</td>
<td>200</td>
<td>274</td>
<td>523</td>
<td>108</td>
<td>76</td>
<td>195</td>
<td>599</td>
<td>158</td>
<td>2</td>
</tr>
</tbody>
</table>

$T(n)$
Understanding $O(n^2)$

\[
T(n) = (n/2)^2 = n^2/4
\]
Understanding $O(n^2)$

<table>
<thead>
<tr>
<th>T(n)</th>
<th>T(1/2n) ≈ 1/4 T(n)</th>
<th>T(1/2n) ≈ 1/4 T(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 14 3 43 200 274 523 108 76 195 599 158 2 260 11 64 932 5</td>
<td>195 599 158 2 260 11 64 932 5</td>
<td>195 599 158 2 260 11 64 932 5</td>
</tr>
</tbody>
</table>

$(n/2)^2 = n^2/4$
Understanding $O(n^2)$

$T(n) = \frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$

$T(\frac{1}{2}n) \approx \frac{1}{4} T(n)$

$(\frac{n}{2})^2 = \frac{n^2}{4}$
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: Merge is linear
Key Insight: **Merge is linear**
Key Insight: **Merge is linear**

Each step makes one comparison and reduces the number of elements to be merged by 1. If there are $n$ total elements to be merged, merging is $\mathcal{O}(n)$.
Divide and Conquer

<table>
<thead>
<tr>
<th></th>
<th>100</th>
<th>14</th>
<th>3</th>
<th>43</th>
<th>200</th>
<th>274</th>
<th>523</th>
<th>108</th>
<th>76</th>
<th>195</th>
<th>599</th>
<th>158</th>
<th>2</th>
<th>260</th>
<th>11</th>
<th>64</th>
<th>932</th>
<th>5</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>T(n)</th>
<th>T((\frac{1}{2}n)) \approx \frac{1}{4} T(n)</th>
<th>T((\frac{1}{2}n)) \approx \frac{1}{4} T(n)</th>
</tr>
</thead>
</table>

14  | 43  | 76  | 100 | 108 | 200 | 274 | 523 | 11  | 64  | 158 | 195 | 260 | 599 | 932 | 5   |
Speed up insertion sort by a factor of two by splitting in half, sorting separately and merging results!
Divide and Conquer

Splitting in two gives 2x improvement.
Divide and Conquer

Splitting in two gives $2x$ improvement.

Splitting in four gives $4x$ improvement.
Divide and Conquer

Splitting in two gives $2x$ improvement.

Splitting in four gives $4x$ improvement.

Splitting in eight gives $8x$ improvement.
Divide and Conquer

Splitting in two gives 2x improvement.

Splitting in four gives 4x improvement.

Splitting in eight gives 8x improvement.

What if we never stop splitting?
Merge Sort
Merge Sort

14  3  43  200  274  523  108  76  195  599  158  2  260  11  64  932

14  3  43  200  274  523  108  76

195  599  158  2  260  11  64  932

14  3  43  200  274  523  108  76

195  599  158  2  260  11  64  932

14  3  43  200  274  523  108  76

195  599  158  2  260  11  64  932

14  3  43  200  274  523  108  76

195  599  158  2  260  11  64  932

14  3  43  200  274  523  108  76

195  599  158  2  260  11  64  932

14  3  43  200  274  523  108  76

195  599  158  2  260  11  64  932

14  3  43  200  274  523  108  76

195  599  158  2  260  11  64  932
Merge Sort

14  3  43  200  274  523  108  76  195  599  158  2  260  11  64  932

14  3  43  200  274  523  108  76
195  599  158  2  260  11  64  932

14  3  43  200
274  523  108  76
195  599  158  2
260  11  64  932

3  14  43  200
274  523  76  108
195  599  2  158
11  26  64  932

14  3  43  200  274  523  108  76  195  599  158  2  260  11  64  932
Merge Sort
Merge Sort
Merge Sort
Merge Sort
Merge Sort Analysis

<table>
<thead>
<tr>
<th></th>
<th>n</th>
<th>n/2</th>
<th>n/4</th>
<th>...</th>
<th>n/2^k</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>11</td>
<td>14</td>
<td>26</td>
<td>43</td>
</tr>
<tr>
<td>64</td>
<td>76</td>
<td>108</td>
<td>158</td>
<td>195</td>
<td>200</td>
</tr>
<tr>
<td>274</td>
<td>523</td>
<td>2</td>
<td>11</td>
<td>26</td>
<td>64</td>
</tr>
<tr>
<td>599</td>
<td>932</td>
<td>3</td>
<td>14</td>
<td>43</td>
<td>26</td>
</tr>
<tr>
<td>932</td>
<td>195</td>
<td>108</td>
<td>76</td>
<td>274</td>
<td>523</td>
</tr>
</tbody>
</table>

Merge how many times?
Merge Sort Analysis

Merge how many times? $n/2^k = 1$

$n = 2^k$

$log_2 n = k$
Merge Sort Analysis

Merge \( n \) elements \( \log_2 n \) times
Merge Sort Analysis

O(n log n)
Merge Sort

How would you code this?
Merge Sort

How would you code this?

Hint: Divide and Conquer!!!
void mergeSort(array)
{
    if array size <= 1
        return //base case
    split array into left_array and right_array
    mergeSort(left_array)
    mergeSort(right_array)
    merge(left_array, right_array, sorted_array)
}
Merge Sort Analysis

Execution time does NOT depend on initial arrangement of data

**Worst Case:** $O(n \log n)$ comparisons and data moves

**Best Case:** $O(n \log n)$ comparisons and data moves

Stable

Best we can do with *comparison-based* sorting that does not rely on a data structure in the worst case => can’t beat $O(n \log n)$

**Space overhead:** auxiliary array at each merge step
What we have so far

<table>
<thead>
<tr>
<th>Sort</th>
<th>Worst Case</th>
<th>Best Case</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Selection Sort</strong></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td><strong>Insertion Sort</strong></td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>Bubble Sort</strong></td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>Merge Sort</strong></td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
Quick Sort
Quick Sort

Select a pivot. Arrange other entries s.t. entries in left partition are \( \leq \) pivot and entries in right partition are \( > \) pivot.
Quick Sort

pivot

<= pivot
> pivot

Partition
Quick Sort

Select a pivot. Arrange other entries s.t. entries in left partition are $\leq$ pivot and entries in right partition are $> pivot$
Quick Sort

Select a pivot. Arrange other entries such that entries in left partition are ≤ pivot and entries in right partition are > pivot.
Quick Sort

Select a pivot. Arrange other entries s.t. entries in left partition are ≤ pivot and entries in right partition are > pivot
Quick Sort

Select a pivot. Arrange other entries s.t. entries in left partition are $\leq$ pivot and entries in right partition are $>$ pivot.
Quick Sort

Select a pivot. Arrange other entries s.t. entries in left partition are $\leq$ pivot and entries in right partition are $>$ pivot
Select a pivot. Arrange other entries s.t. entries in left partition are ≤ pivot and entries in right partition are > pivot.
Quick Sort

Select a pivot. Arrange other entries s.t. entries in left partition are \( \leq \) pivot and entries in right partition are \( > \) pivot.
Quick Sort

Select a pivot. Arrange other entries s.t. entries in left partition are $\leq$ pivot and entries in right partition are $>$ pivot.
Quick Sort

Select a pivot. Arrange other entries s.t. entries in left partition are ≤ pivot and entries in right partition are > pivot.
Quick Sort

Select a pivot. Arrange other entries s.t. entries in left partition are ≤ pivot and entries in right partition are > pivot.
Quick Sort

Select a pivot. Arrange other entries s.t. entries in left partition are ≤ pivot and entries in right partition are > pivot.
Quick Sort

Select a pivot. Arrange other entries s.t. entries in left partition are \( \leq \) pivot and entries in right partition are \( > \) pivot.
Quick Sort

Select a pivot. Arrange other entries s.t. entries in left partition are $\leq$ pivot and entries in right partition are $>$ pivot.
Quick Sort

Select a pivot. Arrange other entries
s.t. entries in left partition are ≤ pivot
and entries in right partition are > pivot
Quick Sort Analysis

Divide and Conquer

\textbf{n comparisons} for each partition

How many subproblems? => Depends on pivot selection

Ideally partition divides problem into two \( \frac{n}{2} \) subproblems for \( \log n \) recursive calls (Best case)

Possibly (though unlikely) each partition has 1 empty subarray for \( n \) recursive calls (Worst case)
template<class T>
void quickSort(T the_array[], int first, int last)
{
    if (last - first + 1 < MIN_SIZE)
    {
        insertionSort(the_array, first, last);
    }
    else
    {
        // Create the partition: S1 | Pivot | S2
        int pivot_index = partition(the_array, first, last);

        // Sort subarrays S1 and S2
        quickSort(the_array, first, pivot_index - 1);
        quickSort(the_array, pivotIndex + 1, last);
    } // end if
} // end quickSort
How to select pivot?
How to select pivot?

Ideally median
Need to sort array to find median

Other ideas?
How to select pivot?

Ideally median
Need to sort array to find median

Other ideas?
Pick first, middle, last position and order them making middle the pivot
How to select pivot?

Ideally median
Need to sort array to find median

Other ideas?
Pick first, middle, last position and order them making middle the pivot
Quick Sort Analysis

Execution time DOES depend on initial arrangement of data AND on PIVOT SELECTION (luck?) => on random data can be faster than Merge Sort

Optimization (e.g. smart pivot selection, speed up base case, iterative instead of recursive implementation) can improve actual runtime -
fastest comparison-based sorting algorithm on average

**Worst Case:** $O(n^2)$ comparisons and data moves

**Best Case:** $O(n \log n)$ comparisons and data moves

Unstable
<table>
<thead>
<tr>
<th>Sort</th>
<th>Worst Case</th>
<th>Best Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$O(n^2)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
https://www.youtube.com/watch?v=kPRA0W1kECg