Sorting
Sorting an array:

\[ [a_1, \ldots, a_n] \rightarrow [a_{\pi(1)}, \ldots, a_{\pi(n)}] \]

such that \( a_{\pi(1)} \leq \ldots \leq a_{\pi(n)} \).

This notation is useful when doing formal analysis of algorithm runtimes.
Known sorts:

<table>
<thead>
<tr>
<th>Sort</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$O(n^2), O(n \log n)$</td>
</tr>
<tr>
<td>Bogo Sort</td>
<td>$O((n + 1)!), O(\infty)$</td>
</tr>
</tbody>
</table>

The second complexity for the last two algorithms is the randomized complexity.
Which suggests that you shouldn’t use Bogo Sort.
Merge sort:

- Break the list into two lists:
  \[ [a_1, \ldots, a_{\lfloor n/2 \rfloor}], [a_{\lfloor n/2 \rfloor}+1, \ldots, a_n] \]

- Sort them recursively using Merge Sort
  \[ [b_1, \ldots, b_{\lfloor n/2 \rfloor}], [b_{\lfloor n/2 \rfloor}+1, \ldots, b_n] \]

- Merge them to get the sorted list
  \[ [b_1, \ldots, b_{\lfloor n/2 \rfloor}] \]
  \[ [a_{\pi(1)}, \ldots, a_{\pi(n)}] \]
  \[ [b_{\lfloor n/2 \rfloor}+1, \ldots, b_n] \]
You can merge in $O(n)$ by using comparing the first elements in each list and choosing the smaller of the two.

Time complexity:

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

Which we showed in the previous class, to obtain

$$T(n) = O(n \log n)$$
You can merge in $O(n)$ by using comparing the first elements in each list and choosing the smaller one of the two.

Time complexity:

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

Which we showed in the previous class, to obtain

$$T(n) = O(n \log n)$$
Counting Inversions:
Given an array \([a_1, \ldots, a_n]\), an inversion is an ordered pair \((a_i, a_j)\) such that \(i < j\) and \(a_i > a_j\).
E.g. in the array 1, 4, 3, 2, 5 the pairs (4, 3), (4, 2) are inversions.
How do we count the number of inversions?
Naive: Check for all pairs of indices.
Time complexity: \(O(n^2)\).
Can we do better?
Quick sort:

- Choose an element $a_i$ as a pivot and move all elements smaller than it to its left and larger elements to its right.
- Sort each side of the list recursively.

\[
[1, 5, 9, 8, 7, 6, 3, 2, 4] \\
\downarrow \\
[1, 5, 6, 3, 2, 4, 7, 9, 8] \\
\downarrow \\
[1, 2, 3, 4, 5, 6, 7, 8, 9]
\]
Time complexity analysis:
In the worst case scenario, we could potentially choose the smallest of the largest element as the pivot, which would give us

\[ T(n) = T(n - 1) + O(n) \]

Which means \( T(n) = O(n^2) \)
Can we somehow make this better?
Can we choose a better pivot?
Quick Select
Problem
Given an array $[a_1, \ldots, a_n]$, find the $k$'th element in the sorted order of the list.

We can use this to find the median, if $k = \frac{n}{2}$. 
Naive algorithm:

- Sort the list
- Find the $k$’th element

But we need the $n/2$’th element to sort...
This is circular...
Naive algorithm:

- Sort the list
- Find the $k$’th element

But we need the $n/2$’th element to sort...
This is circular...
Use merge sort.
Can we do better?
Idea:

- First find an approximate median, \( m \), to split the list.
- Use \( m \) as a pivot to further split the list and find the exact median (or in general the \( k \)’th item).
Median of Medians algorithm:

Divide the array into groups of 5.

\[ a_1, a_2, a_3, a_5, a_5 \]

\[ a_6, a_7, a_8, a_9, a_{10} \]

\[ \vdots \]

\[ a_{n-4}, a_{n-3}, a_{n-2}, a_{n-1}, a_n \]

Find the median of each group, WLOG assume them to be \( a_3, a_8, \ldots, a_{n-2} \).
Find the median of the numbers \( a_3, a_8, \ldots, a_{n-2} \), call it \( m \).
Use \( m \) as a pivot to partition the original array \( a_1, \ldots, a_n \).

\[ \begin{align*}
\underline{\text{smaller than } m = l_1} & \quad m & \quad \underline{\text{larger than } m = l_2}
\end{align*} \]

Now, depending on the size of \( l_1, l_2 \) you can find the the \( k \)'th item in the appropriate list by recursion.
Algorithm 1 Quick Select

function QuickSelect(Int a[], Int k)
    medians ← [median($a_1, \ldots, a_5$), \ldots,median($a_{n-4}, \ldots, a_n$)]
    m ← QuickSelect(medians, $\left\lfloor \frac{|\text{medians}|}{2} \right\rfloor$)
    less_m ← [x | x ∈ a and x ≤ m]
    more_m ← [x | x ∈ a and x > m]
    if |less_m| ≥ k then
        return QuickSelect(less_m, k)
    else
        return QuickSelect(more_m, k - |less_m|)
    end if
end function
We want to show that, while $m$ is still not the median of $a_1, \ldots, a_n$ but it performs a better split than a random pivot
\[ |l_1| \leq n - |S_2|, \quad l_2 \leq n - |S_1| \]

\[ |S_1| = |S_2| = \frac{3n}{10} \]

\[ \implies l_1, l_2 \leq \frac{7n}{10} \]

So now the sizes are a fraction of the original list.
Time complexity analysis:

\[ T(n) = \begin{array}{c} T \left( \frac{n}{5} \right) \\ \text{median of medians} \\ T \left( \frac{7n}{10} \right) \\ \text{max recursion call} \\ O(n) \\ \text{partitioning} \end{array} \]
Time complexity analysis:

\[ T(n) = \left\{ \begin{array}{ll}
T \left( \frac{n}{5} \right) & \text{median of medians} \\
T \left( \frac{7n}{10} \right) & \text{max recursion call} \\
O(n) & \text{partitioning}
\end{array} \right. \]

Written assignment: Prove that \( T(n) = O(n) \)
Lower Bounds
In general proving lower bounds is hard problem.
This is the essence of one of the Clay millennium problems: P vs NP.
In general proving lower bounds is hard problem. This is the essence of one of the Clay millennium problems: P vs NP. But some easier lower bounds can be proved for simple algorithms, such as sorting, convex hull, closest pair of points.
Sorting:
Any comparison based sorting technique is $\Omega(n \log n)$. 
Sorting:
Any comparison based sorting technique is \( \Omega(n \log n) \).
For simplicity assume that we want to sort the numbers 1, \ldots, n.
We can only perform queries of the form $a_i \geq a_j$. Depending on the answer to this query, our algorithm will decide its next step.

This is called the decision tree model of computation.
Points to note:

- Leaf nodes represent the point where we have sorted the array.
- Two different permutations cannot lead to the same leaf.

Proof: Let the input permutations sorted order be $a_{\pi(1)}, \ldots, a_{\pi(n)}$. Then we must have compared $a_{\pi(i)}, a_{\pi(i+1)}$ at some point. Because if we did not compare these two elements, we could swap their positions in the original list and all other comparisons would still give the same result, leading to the same leaf, but the output would not be sorted.

And given all comparisons $(a_i, a_{i+1})$, we can now sort the input easily.

Based on this, we have at least $n!$ leaves in our tree, each corresponding to one permutation of $1, \ldots, n$. 
If the height of the tree is \( h \), then there are \( 2^h \) leaves in the tree.

\[
\implies 2^h \geq n! \\
h \geq \log n! \\
h \geq \log \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \quad \text{Stirling’s formula} \\
h = \Omega(n \log n)
\]

Hence, any comparison based sorting algorithm has a path of at least \( O(n \log n) \) in its decision tree, giving us the desired lower bound.
Lower bound for Convex Hull problem: $\Omega(n \log n)$.
We can use the lower bound for sorting to prove a lower bound for the convex hull problem.
This is called reducing sorting to convex hull.

\[
\begin{align*}
\text{Sorting} & \quad \rightarrow \quad \text{Convex Hull} \\
\downarrow & \\
\text{Sorting Solution} & \quad \leftarrow \quad \text{Convex Hull Solution}
\end{align*}
\]
Input for sorting: $a_1, \ldots, a_n$.
Convert it to an input for convex hull:

$$(a_1, a_1^2), (a_2, a_2^2), \ldots, (a_n, a_n^2)$$

Find the convex hull for these points. The points lie on a parabola, hence it will be of the format:
The output of the convex hull gives the sorted order of the points.

\[ T_{\text{sort}}(n) \leq T_{\text{cnv}}(n) + O(n) \]

We have already show that sorting is lower bounded by \( \Omega(n \log n) \), which would be contradicted if the convex hull problems has a faster solution.

Hence, convex hull is also lower bounded by \( \Omega(n \log n) \).
To prove the lower bound of the closest pair of points problem, we use another problem, the Element Uniqueness problem.

Problem
Given an array of integers, $a_1, \ldots, a_n$, print YES if all of them are unique, else print NO.

Claim: Element Uniqueness problem is lower bounded by $\Omega(n \log n)$. 
We again use the decision tree computation model for the EU problem. Each leaf is labeled either YES or NO depending on the fact that the array has no repetitions and if it does, respectively. If the input is \([a_1, \ldots, a_n]\) and the sorted order of this is \([a_{\pi(1)}, \ldots, a_{\pi(n)}]\), then on any path to a leaf node, we must have compared \(a_{\pi(i)}, a_{\pi(i+1)}\). If we did not compare them, then we can replace \(a_{\pi(i+1)}\) with \(a_{\pi(i)}\), which does have a duplicate and still goes to the same leaf, which is a contradiction.
We again use the decision tree computation model for the EU problem.
Each leaf is labeled either YES or NO depending on the fact that the array has no repetitions and if it does, respectively.
If the input is \([a_1, \ldots, a_n]\) and the sorted order of this is \([a_{\pi(1)}, \ldots, a_{\pi(n)}]\), then on any path to a leaf node, we must have compared \(a_{\pi(i)}, a_{\pi(i+1)}\).
If we did not compare them, then we can replace \(a_{\pi(i+1)}\) with \(a_{\pi(i)}\), which does have a duplicate and still goes to the same leaf, which is a contradiction.

Now the same analysis for height gives us the desired lowerbound.
Input for EU: \([a_1, \ldots, a_n]\).
Convert it to input for Closest Pair: \((a_1, a_1), (a_2, a_2), \ldots, (a_n, a_n)\). Now find the closest distance between points.
If distance is 0 print NO. Else print YES.
Input for EU: \([a_1, \ldots, a_n]\).
Convert it to input for Closest Pair: \((a_1, a_1), (a_2, a_2), \ldots, (a_n, a_n)\). Now find the closest distance between points.
If distance is 0 print NO. Else print YES.
This shows that if we can solve the closest pair of points faster than \(O(n \log n)\) we can solve EU faster as well.
Hence, Closest Pair is lower bounded by \(\Omega(n \log n)\).