The best sorting algorithm that sorts by comparing the elements with each other is order $N \log(N)$. Why?
The reason is a consequence of properties of binary trees (not binary search trees).

- In the argument we assume the keys are distinct, this means for each comparison there are only 2 possibilities.
- For any sorting algorithm, each execution consists of a sequence of comparisons until the input is sorted.
- We want to know how many comparisons are needed in the worst case.
- We can represent the comparisons made in any execution by a binary tree – a comparison tree.
  - The root is the first comparison
  - For each comparison except the last, its left child is the next comparison if the result was “less” and its right child is the next comparison if the result was “greater”
  - For a last comparison, the children are the sorted result orders.
Bubble Sort for input size 3: $x$ $y$ $z$

**Bubble Sort** $x$ $y$ $z$:

- Compare $x$ and $y$. If $x > y$: $y$ $x$ $z$
- Compare $x$ and $z$. If $x > z$: $y$ $z$ $x$
- Compare $y$ and $z$. If $y > z$: $z$ $y$ $x$

This gives one path in the comparison tree from the root to a leaf.

Leaves are the sorted results of the comparisons along each path from the root.

How many leaves must there be? For $N = 3$, there are 6 possible orderings of the inputs $x$ $y$ $z$:

1. $x < y < z$, 2. $x < z < y$, 3. $y < x < z$, 4. $y < z < x$, 5. $z < x < y$, 6. $z < y < x$

The comparison tree must have each one of these 6 as a sorted result (= a leaf).
How many leaves must there be any comparison tree to sort $N$ distinct input values?
Answer: $N!$ One for each of the possible input orders of $N$ distinct items.

How many comparisons are needed by any sorting algorithm that sorts by comparing input items in the worst case?
• Any such sorting algorithm execution determines a binary comparison tree.
• The number of comparisons made in the worst case is the height of the comparison tree.
Claim: If a binary tree of height $h$ has $M$ nodes, then
\[ M \leq 2^{h+1} - 1 \]

The maximum number of nodes in a binary tree of height $h$:

<table>
<thead>
<tr>
<th>Height</th>
<th>Maximum number of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 = $2^1 - 1$</td>
</tr>
<tr>
<td>1</td>
<td>3 = $2^2 - 1$</td>
</tr>
<tr>
<td>2</td>
<td>7 = $2^3 - 1$</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$h$</td>
<td>$2^{h+1} - 1$</td>
</tr>
</tbody>
</table>
Claim: If every node is either a leaf or has two children, then 
#leaves = #internal nodes + 1 and 
Total #nodes = 2 * #leaves – 1

An internal node is a node with 2 children

L = #leaves
T = #internal nodes
M = total #nodes

1. For each internal node add 2 to a node count; one for its left child and one for its right child. So this count is 2T.
2. 2T counts every node except the root since the root is not a child of any node.
3. So M = 2T + 1
4. But since every node is a leaf or an internal node, M = L + T
5. So L + T = 2T + 1
6. L = T + 1 and T = L – 1
7. M = L + T = L + L – 1 = 2L - 1
Claim: For a comparison tree for sorting N items the height h must be at least \( \log(N!) \). That is,

\[
\log(N!) \leq h
\]

L = #leaves
M = total #nodes (internal and leaves)
1. If a binary tree of height h has M nodes, then
   \[
   M \leq 2^{h+1} - 1
   \]
2. \( M = 2L - 1 \) (since for a comparison tree every node is a leaf or has 2 children)
3. \( 2L = M + 1 \leq 2^{h+1} \)
4. Applying log to 3 gives
   \[
   \log(2L) \leq h + 1
   \]
5. \( \log(L) \leq h \) by 4 since \( \log(2L) = \log(2) + \log(L) = 1 + \log(L) \)
6. \( L = N! \) for a comparison tree
7. \( \log(N!) \leq h \)

So the height is at least \( \log(N!) \), but what order of growth is that?
Claim: \( \log(N!) \) is order at least \( N\log(N) \)

\[
\log(N!) = \log(1) + \log(2) + \log(3) + \ldots + \log(N) \\
\geq \log\left(\frac{N}{2}\right) + \ldots + \log(N) \\
\geq \log\left(\frac{N}{2}\right) + \log\left(\frac{N}{2}\right) + \ldots + \log\left(\frac{N}{2}\right) \\
\geq \frac{N}{2} \log\left(\frac{N}{2}\right) = 0.5N\log(N) - 0.5N
\]

Conclusion: The worst number of comparisons that any comparison based sorting algorithm makes has order of growth at least

\( N\log(N) \)
The best sorting algorithm is $N \log(N)$, but this is counting *comparisons* between elements.

Some sorting methods don't use any element comparisons! One such method uses "key index counting".
Consider sorting the 3 digit numbers:

412
012
101
432
302
201
410

The idea to make 3 passes (in this case, since these are 3 digit numbers) starting with the least significant digit (LSD) on the first pass.

After the k-th pass, the elements will be in sorted order if only the last k digits are considered.
On the k-th pass count the number of occurrences of each digit in position k (starting with the least significant digit; digit positions are numbered right to left starting at k = 0). After the k-th pass,

\[
\text{count}[d + 1] = \text{the number of occurrences of digit } d \text{ in position } k
\]

So after pass k = 0,

\[
\begin{align*}
\text{count}[1] &= \text{number of occurrences of digit 0} \\
\text{count}[2] &= \text{number of occurrences of digit 1} \\
\text{count}[3] &= \text{number of occurrences of digit 2} \\
&\vdots
\end{align*}
\]

This means count[0] will always be 0.
After pass $k = 0$, the count array is:

<table>
<thead>
<tr>
<th>a</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 412</td>
<td>0 0</td>
</tr>
<tr>
<td>1 012</td>
<td>1 1</td>
</tr>
<tr>
<td>2 101</td>
<td>2 2</td>
</tr>
<tr>
<td>3 432</td>
<td>3 4</td>
</tr>
<tr>
<td>4 302</td>
<td>4 0</td>
</tr>
<tr>
<td>5 201</td>
<td>5 0</td>
</tr>
<tr>
<td>6 410</td>
<td>6 0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9 0</td>
<td></td>
</tr>
</tbody>
</table>
**Compute the Index**

After calculating the frequencies of each digit in position $k$ and storing in the count array, update count so that it provides instead the **new index** where each value belongs so that the values are sorted by the digit in position $k$. The idea for transforming the count array to hold the index position is:

**Idea:** The new **index** of a key is the **number** of preceding keys whose $k$-th digit is less than or equal to the $k$-th digit of this key.
Compute the Index

The count array is updated so that count[d] is the index in the array where the first number with digit d should go in sorted order for the position for this pass.

\[
\text{count}[0] = 0
\]
and for \( d > 0 \)

\[
\text{count}[d] = \text{count}[d] + \text{count}[d - 1]
\]

Note that index 7 is out of bounds. But for this pass the position is the least significant digit (position 0) and only the digits 0, 1, and 2 occur.
Use the Index to insert the items in an auxillary array

Copy the values from the array a to an auxilliary array, aux, at the index indicated by the count array and update that index for the digit position for this pass.

<table>
<thead>
<tr>
<th>a</th>
<th>count</th>
<th>aux</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 412</td>
<td>0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 012</td>
<td>1 1 1</td>
<td>1</td>
</tr>
<tr>
<td>2 101</td>
<td>2 3 2</td>
<td>2</td>
</tr>
<tr>
<td>3 432</td>
<td>3 7 3</td>
<td>3 412</td>
</tr>
<tr>
<td>4 302</td>
<td>4 7 4</td>
<td>4</td>
</tr>
<tr>
<td>5 201</td>
<td>5 7 5</td>
<td>5</td>
</tr>
<tr>
<td>6 410</td>
<td>6 7 6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>9 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

412 is copied to the position indicated by count[2], which is position 3. count[2] should now be incremented to 4, to be ready for the next element whose digit is 2.
Use the Index to insert the items in an auxillary array

Copy the values from the array a to an auxilliary array, aux, at the index indicated by the count array and update that index for the digit position for this pass.

<table>
<thead>
<tr>
<th>a</th>
<th>count</th>
<th>aux</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 412</td>
<td>0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 012</td>
<td>1 1 1</td>
<td>1</td>
</tr>
<tr>
<td>2 101</td>
<td>2 3 4</td>
<td>2</td>
</tr>
<tr>
<td>3 432</td>
<td>3 7 3 412</td>
<td></td>
</tr>
<tr>
<td>4 302</td>
<td>4 7 4</td>
<td></td>
</tr>
<tr>
<td>5 201</td>
<td>5 7 5</td>
<td></td>
</tr>
<tr>
<td>6 410</td>
<td>6 7 6</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>9 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

412 is copied to the position indicated by count[2], which is position 3. count[2] should now be incremented to 4, to be ready for the next element whose digit is 2.
After inserting the next 2 elements

Copy the values from the array a to an auxilliary array, aux, at the index indicated by the count array and update that index for the digit position for this pass.

\[
\begin{array}{cccc}
\text{a} & \text{count} & \text{aux} \\
0 & 412 & 0 & 0 \\
1 & 012 & 1 & 1 \\
2 & 101 & 2 & 2 \\
3 & 432 & 3 & 3 \\
4 & 302 & 4 & 4 \\
5 & 201 & 5 & 5 \\
6 & 410 & 6 & 6 \\
\ldots & \ldots & \ldots & \ldots \\
9 & 43 & 2 & 2
\end{array}
\]

432 will be copied to the position indicated by count[2], which is now index position 5. Similarly, when 201 is processed it will be copied to position in count[1], which is now position 2.
Result after insertion step of the first pass

Copy the values from the array $a$ to an auxilliary array, $aux$, at the index indicated by the count array and update that index for the digit position for this pass.

<table>
<thead>
<tr>
<th>a</th>
<th>count</th>
<th>aux</th>
<th>(index) updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 412</td>
<td>0</td>
<td></td>
<td>0 410</td>
</tr>
<tr>
<td>1 012</td>
<td>1</td>
<td>4 2 3</td>
<td>1 101</td>
</tr>
<tr>
<td>2 101</td>
<td>2</td>
<td>3 4 5 6 7</td>
<td>2 201</td>
</tr>
<tr>
<td>3 432</td>
<td>3</td>
<td>7</td>
<td>3 412</td>
</tr>
<tr>
<td>4 302</td>
<td>4</td>
<td>7</td>
<td>4 012</td>
</tr>
<tr>
<td>5 201</td>
<td>5</td>
<td>7</td>
<td>5 432</td>
</tr>
<tr>
<td>6 410</td>
<td>6</td>
<td>7</td>
<td>6 302</td>
</tr>
</tbody>
</table>

... 

... 

9 7
Final Step for pass 0: Exchange aux and a

To get ready for the second pass, exchange arrays aux and a. We don’t have to copy the values, just exchange the references, but the count array should be reinitialized to 0.

<table>
<thead>
<tr>
<th>a</th>
<th>count</th>
<th>aux</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 410</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 101</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 201</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3 412</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4 012</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5 432</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6 302</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

...
Second Pass (position k = 1) after count step and index step:

The first step of this pass counted the frequencies of the digits in the position 1 (second from the right) and the second step updated count to be hold the indices. The third step is just starting with the first value, 410, copied to aux.

<table>
<thead>
<tr>
<th>(freq)</th>
<th>(index)</th>
</tr>
</thead>
<tbody>
<tr>
<td>old</td>
<td>updated</td>
</tr>
<tr>
<td>a</td>
<td>count</td>
</tr>
<tr>
<td>0 410</td>
<td>0 0</td>
</tr>
<tr>
<td>1 101</td>
<td>1 3</td>
</tr>
<tr>
<td>2 201</td>
<td>2 3</td>
</tr>
<tr>
<td>3 412</td>
<td>3 0</td>
</tr>
<tr>
<td>4 012</td>
<td>4 1</td>
</tr>
<tr>
<td>5 432</td>
<td>5 0</td>
</tr>
<tr>
<td>6 302</td>
<td>6 0</td>
</tr>
</tbody>
</table>

...
Sorting Strings with LSD

Instead of 10 digits, LSD uses 256 possible characters for each position of Strings.

It is appropriate for sorting strings *that are all the same length*; that is, for *strings of fixed length*.

A bigger the character set, will require a bigger count array. LSD uses on the order of

\[ 7WN + 3WR \]

array accesses where

W is the (fixed) length of the strings
N is the number of strings to sort
R is the number of characters in the character set.
Sorting Social Security Numbers (as Strings)

For N social security numbers without dashes
How many array accesses?

\[ W = 9 \quad (\text{length}) \]
\[ R = ? \quad (\text{character set size}) \]

So

\[ 7WN + 3WR = 63N + 270 \quad (\text{for } R = 10; \text{ just the digits 0 - 9}) \]
\[ 7WN + 3WR = 63N + 6912 \quad (\text{for } R = 256; \text{ all 256 ascii characters}) \]

Is LSD faster than Java's Arrays.sort for Strings?
Sorting Variable Length Strings

LSD requires *fixed length strings.*

MSD (most significant 'digit') sorts *variable length strings* by considering the most significant character (left most) first.

Read section 4.1 for LSD details and for MSD to be discussed next time.

But is MSD faster than Arrays.sort for Strings?