CSE 12: Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Sixteen
1 August 2012
Sorting, continued
Review of selection sort

• Pseudocode:

```java
void selectionSort (int[] array) {
    While size of unsorted part > 0:
        Find largest element e of unsorted part
        Swap e with right-most element of unsorted part
}
```

• Time cost: $O(n^2)$ in worst, best, and average cases

• No matter what the input array contains, selection sort must always search the unsorted part for its largest remaining element.
Selection sort: in-place

- Example:

Unsorted part   Sorted part
6  1  4  3  8  7  2  5
Selection sort: in-place

- Example:

Unsorted part  Sorted part
6  1  4  3  5  7  2  8
Selection sort: in-place

- Example:

<table>
<thead>
<tr>
<th>Unsorted part</th>
<th>Sorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 1 4 3 5 2</td>
<td>7 8</td>
</tr>
</tbody>
</table>
Selection sort: in-place

- Example:

Unsorted part  Sorted part
2  1  4  3  5  6  7  8
Selection sort: in-place

- Example:

  Unsorted part  Sorted part

  1  2  3  4  5  6  7  8

Wednesday, August 1, 12
Insertion sort.
Insertion sort

- Like selection sort, **insertion sort** maintains a “sorted part” $S$ and “unsorted part” $U$ of the input array:

```
Sorted part     Unsorted part
```

- Insertion sort operates by repeatedly removing the *leftmost* element of $U$ and *inserting* it into its “proper place” in $S$. 
Insertion sort

- Example:

  Sorted part

  Unsorted part

  6 1 4 3 8 7 2 5
Insertion sort

• Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1 4 3 8 7 2 5</td>
</tr>
<tr>
<td>6</td>
<td>1 4 3 8 7 2 5</td>
</tr>
</tbody>
</table>
Insertion sort

- Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1 4 3 8 7 2 5</td>
</tr>
</tbody>
</table>

"Wednesday, August 1, 12"
Insertion sort

- Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 1 4 3 8 7 2 5</td>
<td>6 1 4 3 8 7 2 5</td>
</tr>
<tr>
<td>6 1 4 3 8 7 2 5</td>
<td>6 1 4 3 8 7 2 5</td>
</tr>
<tr>
<td>6 1 4 3 8 7 2 5</td>
<td>6 1 4 3 8 7 2 5</td>
</tr>
</tbody>
</table>
Insertion sort

- Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1 4 3 8 7 2 5</td>
</tr>
<tr>
<td>1 6</td>
<td>1 4 3 8 7 2 5</td>
</tr>
<tr>
<td></td>
<td>4 3 8 7 2 5</td>
</tr>
</tbody>
</table>
Insertion sort

- Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1 4 3 8 7 2 5</td>
</tr>
<tr>
<td>1 6</td>
<td></td>
</tr>
<tr>
<td>1 4 6</td>
<td></td>
</tr>
</tbody>
</table>
Insertion sort

- Example:

  Sorted part

  6
  1 6
  1 4 6

  Unsorted part

  6 1 4 3 8 7 2 5
  1 4 3 8 7 2 5
  4 3 8 7 2 5
  3 8 7 2 5
Insertion sort

- Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1 4 3 8 7 2 5</td>
</tr>
<tr>
<td>1 6</td>
<td>1 4 3 8 7 2 5</td>
</tr>
<tr>
<td>1 4 6</td>
<td>4 3 8 7 2 5</td>
</tr>
<tr>
<td>1 3 4 6</td>
<td>3 8 7 2 5</td>
</tr>
<tr>
<td></td>
<td>8 7 2 5</td>
</tr>
</tbody>
</table>

Wednesday, August 1, 12
**Insertion sort**

- **Example:**

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1 4 3 8 7 2 5</td>
</tr>
<tr>
<td>1 6</td>
<td></td>
</tr>
<tr>
<td>1 4 6</td>
<td></td>
</tr>
<tr>
<td>1 3 4 6</td>
<td></td>
</tr>
</tbody>
</table>
Insertion sort

• Example:

Sorted part

Unsorted part

6
1 6
1 4 6
1 3 4 6
1 3 4 6 8

6 1 4 3 8 7 2 5
1 4 3 8 7 2 5
4 3 8 7 2 5
3 8 7 2 5
8 7 2 5
7 2 5
Insertion sort

- Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1 4 3 8 7 2 5</td>
</tr>
<tr>
<td>6 1</td>
<td>4 3 8 7 2 5</td>
</tr>
<tr>
<td>6 1 4</td>
<td>3 8 7 2 5</td>
</tr>
<tr>
<td>6 1 4 3</td>
<td>8 7 2 5</td>
</tr>
<tr>
<td>6 1 3 4</td>
<td>6 8</td>
</tr>
<tr>
<td>6 1 3 4 6</td>
<td>7 8</td>
</tr>
<tr>
<td>6 1 3 4 6 8</td>
<td></td>
</tr>
<tr>
<td>6 1 3 4 6 7 8</td>
<td></td>
</tr>
<tr>
<td>6 1 2 3 4 6 7 8</td>
<td></td>
</tr>
<tr>
<td>6 1 2 3 4 5 6 7 8</td>
<td>Done.</td>
</tr>
</tbody>
</table>

Done.
Insertion sort

- With insertion sort, most of the “effort” is in *inserting* the element into its proper slot.

- In contrast, with *selection sort*, most of the effort is in *finding* the largest element to insert.

- Like selection sort, insertion sort too can operate *in-place*:
  - When we remove the leftmost element $x$ from $U$, we save it in a temporary variable.
  - To find $x$’s proper “slot”: we “slide down” each element $y$ of $S$ to the right as long as $y > x$.
  - We finally insert $x$, and repeat until $U$ is empty.
Insertion sort

- Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 1 4 3 8 7 2 5</td>
<td>x: 6</td>
</tr>
</tbody>
</table>

Wednesday, August 1, 12
• Example:

Sorted part       Unsorted part
6 1 4 3 8 7 2 5
Insertion sort

• Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 1 4 3 8 7 2 5</td>
<td>x:</td>
</tr>
</tbody>
</table>
Insertion sort

• Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 4 3 8 7 2 5</td>
<td>x: 1</td>
</tr>
</tbody>
</table>

Move $y=6$ to the right because $y>x$. 
Insertion sort

- Example:

  Sorted part  Unsorted part

  6  4  3  8  7  2  5  x: 1
Insertion sort

- Example:

 Sorted part     Unsorted part
     1   6   4   3   8   7   2   5
Insertion sort

- Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 6 4 3 8 7 2 5</td>
<td>x: 4</td>
</tr>
</tbody>
</table>

Wednesday, August 1, 12
Insertion sort

• Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 6</td>
<td>3 8 7 2 5</td>
</tr>
</tbody>
</table>

x: 4

Move y=6 to the right because y>x.
Insertion sort

- Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
<th>x: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 6 3 8 7 2 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Insertion sort**

- **Example:**

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 6 3 8 7 2 5</td>
<td></td>
</tr>
</tbody>
</table>
Insertion sort

• Example:

Sorted part       Unsorted part
1  4  6  3  8  7  2  5

x: 3
### Insertion sort

- **Example:**

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 6</td>
<td>8 7 2 5</td>
</tr>
</tbody>
</table>

Move $y=6$ to the right because $y>x$. 

x: 3
Insertion sort

• Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 6 8 7 2 5</td>
<td>x: 3</td>
</tr>
</tbody>
</table>
Insertion sort

• Example:

Sorted part  Unsorted part
1 4 6 8 7 2 5  x: 3

Move 4 to the right because 4>x.
Insertion sort

- Example:

  Sorted part       Unsorted part
  1  4  6  8  7  2  5              x: 3
Insertion sort

• Example:

Sorted part        Unsorted part
1  3  4  6  8  7  2  5
Insertion sort

• Example:

Sorted part  Unsorted part
1  3  4  6  8  7  2  5
Insertion sort

• Example:

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 4 6 7 8</td>
<td>2 5</td>
</tr>
</tbody>
</table>
Insertion sort

• Example:

Sorted part         Unsorted part
1 2 3 4 6 7 8 5
Insertion sort

- Example:

Sorted part    Unsorted part
1 2 3 4 5 6 7 8
Insertion sort

• Pseudocode:

While U is not empty:
  Save leftmost element x of U into temporary variable.
  Remove x from U.
  Loop from right to left on element y of S:
    If y > x:
      Slide y to the right by one slot.
      Let y be the element to the left of y
    Else (y ≤ x):
      Insert x to the right of y.
    Break

The reason we need this variable is that “sliding” y to the right may overwrite the leftmost element of U.

Since the algorithm requires only $O(1)$ additional memory (to store x), it is still considered to operate “in-place”.

Wednesday, August 1, 12
Stability

• Insertion sort is *stable* as long as we “shift over” an element y in S if $y > x$.

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3₁ 3₂ 7 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3₁ 3₂ 7 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3₁ 3₂ 7 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3₁ 3₂ 7 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3₁ 7 8</td>
<td>x: 3₂</td>
</tr>
</tbody>
</table>

We don’t move $y=3₁$ to the right because $y \text{ not} > x$. 
Stability

- Insertion sort is \textit{stable} as long as we “shift over” an element \( y \) in \( S \) if \( y > x \).

\begin{tabular}{ll}
Sorted part & Unsorted part \\
1 2 3 & 1 3 2 7 8 \\
1 2 3 & 1 3 2 7 8 \\
1 2 3 & 1 3 2 7 8 \\
1 2 3 & 1 3 2 7 8 \\
1 2 3 & 1 3 2 7 8 \\
1 2 3 & 1 3 2 7 8 \\
1 2 3 & 1 3 2 7 8 \\
1 2 3 & 1 3 2 7 8 \\
1 2 3 & 1 3 2 7 8 \\
\end{tabular}
Stability

- Insertion sort is *stable* as long as we “shift over” an element $y$ in $S$ if $y > x$.

1 2 3 3 2 7 8
1 2 3 3 2 7 8
1 2 3 3 2 7 8
1 2 3 3 2 7 8
1 2 3 3 2 7 8
1 2 3 3 2 7 8
1 2 3 3 2 7 8
1 2 3 3 2 7 8
1 2 3 3 2 7 8
Stable.
Stability

• If instead we “shift over” $y$ whenever $y \geq x$, then insertion sort is not stable.

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 $\underline{3}$ 7 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3 $\underline{3}$ 7 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3 $\underline{3}$ 7 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3 $\underline{3}$ 7 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3 $\underline{7}$ 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3 $\underline{7}$ 8</td>
<td>$x: \underline{3}$</td>
</tr>
</tbody>
</table>

We move $y=3$ to the right because $y \geq x$. 

Wednesday, August 1, 12
Stability

• If instead we “shift over” \( y \) whenever \( y \geq x \), then insertion sort is not stable.

Sorted part   Unsorted part

\[
\begin{array}{cccc}
1 & 2 & 3 & 3_2 \ 7 \ 8 \\
1 & 2 & 3 & 3_2 \ 7 \ 8 \\
1 & 2 & 3 & 3_2 \ 7 \ 8 \\
1 & 2 & 3 & 3_2 \ 7 \ 8 \\
1 & 2 & 3 & 3_2 \ 7 \ 8 \\
1 & 2 & 3_1 & 7 \ 8 \\
\end{array}
\]

x: 3_2
Stability

• If instead we “shift over” $y$ whenever $y \geq x$, then insertion sort is not stable.

<table>
<thead>
<tr>
<th>Sorted part</th>
<th>Unsorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3₁ 3₂ 7 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3₁ 3₂ 7 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3₁ 3₂ 7 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3₁ 3₂ 7 8</td>
<td></td>
</tr>
<tr>
<td>1 2 3₂ 3₁ 7 8</td>
<td></td>
</tr>
</tbody>
</table>
Stability

- If instead we “shift over” \( y \) whenever \( y \geq x \), then insertion sort is \textit{not} stable.

\begin{align*}
\text{Sorted part} & \quad \text{Unsorted part} \\
1 & \quad 2 3_1 3_2 7 8 \\
1 & \quad 2 3_1 3_2 7 8 \\
1 & \quad 2 3_1 3_2 7 8 \\
1 & \quad 2 3_1 3_2 7 8 \\
1 & \quad 2 3_1 3_2 7 8 \\
1 & \quad 2 3_2 3_1 7 8 \\
1 & \quad 2 3_2 3_1 7 8 \\
1 & \quad 2 3_2 3_1 7 8 \\
1 & \quad 2 3_2 3_1 7 8 \\
\end{align*}

Not stable.
Time cost: worst case

While U is not empty:
    Save leftmost element x of U into temporary variable.
    Remove x from U.
    Loop from right to left on element y of S:
      If y > x:
        Slide y to the right by one slot.
        Let y be the element to the left of y
      Else (y ≤ x):
        Insert x to the right of y.
        Break

- Outer loop executes n times.
- Inner loop has to move all the elements of S to the right by one slot before inserting x.

- Since S grows in size as outer loop iterates, this results in 1, 2, 3, ..., n-1 operations.
- $1 + 2 + 3 + ... + n-1 = n(n-1)/2 = O(n^2)$.
- Worst case is realized when the input are sorted backwards.
Time cost: best case

While U is not empty:
  Save leftmost element x of U into temporary variable.
  Remove x from U.
  Loop from right to left on element y of S:
    If y > x:
      Slide y to the right by one slot.
      Let y be the element to the left of y
    Else (y ≤ x):
      Insert x to the right of y.
    Break

• Outer loop executes $n$ times.

• Inner loop only executes once -- x is inserted as the rightmost element of $S$.

• This results in only 1 operation per outer loop iteration.

• $1 + 1 + 1 + ... + 1 = O(n)$.

• Best case is realized when the input is already sorted.
Insertion sort versus selection sort

- Selection sort is $O(n^2)$ regardless of the input.
- Insertion sort is $O(n^2)$ in the worst case and average case, but $O(n)$ in the best case.
- If your input array might already be sorted (or almost sorted), then insertion sort is much better.
Heapsort.
Heapsort

- The heap data structure we covered earlier in the course turns out to be useful for sorting.

- A heap allows the removal of the largest element in $O(\log n)$ time.

- To see how this is useful in sorting, recall how selection sort operates:

  While U is not empty:
  
  Remove the largest element of U and add it to S.
Heapsort

• Selection sort uses a simple linear search through $U$ to find the largest element in $O(n)$ time.
• Using a heap, we can do this in $O(\log n)$ time.
• This results in the following heapsort algorithm:

  Build a heap from the data in $U$.
  While $U$ is not empty:
    Remove largest from $U$ and add it to $S$. 
Heapsort

• Building a heap from $n$ data in $U$ takes time at most $O(n \log n)$. *

• The loop iterates $n$ times.
  • Finding+removing largest takes time $O(\log n)$.

• In total, heapsort takes time $O(n \log n) + n*O(\log n) = O(n \log n)$ in both the worst case and best case.

* It’s actually possible to heapify an array of $n$ elements in $O(n)$ time, but that doesn’t affect heapsort’s asymptotic performance.
Example:

Unsorted part

\[6 \ 1 \ 4 \ 3 \ 8 \ 7 \ 2 \ 5\]

Sorted part

First, convert this into an array-based max-heap.
Heapsort

- Example:

Unsorted part

6 1 4 3 8 7 2 5
8 6 7 5 3 4 2 1

Sorted part
Heapsort

- Example:

<table>
<thead>
<tr>
<th>Unsorted part</th>
<th>Sorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 1 4 3 8 7 2 5</td>
<td></td>
</tr>
<tr>
<td>8 6 7 5 3 4 2 1</td>
<td></td>
</tr>
</tbody>
</table>

Now, repeatedly call `removeLargest()` and add that element to the sorted part.
Heapsort

- Example:

Unsorted part

\[
\begin{align*}
6 & \quad 1 & \quad 4 & \quad 3 & \quad 8 & \quad 7 & \quad 2 & \quad 5 \\
8 & \quad 6 & \quad 7 & \quad 5 & \quad 3 & \quad 4 & \quad 2 & \quad 1 \\
7 & \quad 6 & \quad 4 & \quad 5 & \quad 3 & \quad 1 & \quad 2 & \quad 8
\end{align*}
\]

Sorted part

Done.
Heapsort

• Example:

Unsorted part

6 1 4 3 8 7 2 5
8 6 7 5 3 4 2 1
7 6 4 5 3 1 2
6 5 4 2 3 1

Sorted part

8
7 8

Done.
Heapsort

Example:

<table>
<thead>
<tr>
<th>Unsorted part</th>
<th>Sorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 1 4 3 8 7 2 5</td>
<td></td>
</tr>
<tr>
<td>8 6 7 5 3 4 2 1</td>
<td></td>
</tr>
<tr>
<td>7 6 4 5 3 1 2</td>
<td>8</td>
</tr>
<tr>
<td>6 5 4 2 3 1</td>
<td>7 8</td>
</tr>
<tr>
<td>5 3 4 2 1</td>
<td>6 7 8</td>
</tr>
</tbody>
</table>

Done.
Heapsort

• Example:

Unsorted part

Sorted part

6 1 4 3 8 7 2 5
8 6 7 5 3 4 2 1
7 6 4 5 3 1 2
6 5 4 2 3 1
5 3 4 2 1
4 3 1 2

Done.
Heapsort

- Example:

<table>
<thead>
<tr>
<th>Unsorted part</th>
<th>Sorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 1 4 3 8 7 2 5</td>
<td>8</td>
</tr>
<tr>
<td>8 6 7 5 3 4 2 1</td>
<td>6 7 8</td>
</tr>
<tr>
<td>7 6 4 5 3 1 2</td>
<td>5 6 7 8</td>
</tr>
<tr>
<td>6 5 4 2 3 1</td>
<td>4 5 6 7 8</td>
</tr>
<tr>
<td>5 3 4 2 1</td>
<td>3 4 5 6 7 8</td>
</tr>
<tr>
<td>4 3 1 2</td>
<td>2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>3 2 1</td>
<td>1</td>
</tr>
<tr>
<td>2 1</td>
<td>2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>1</td>
<td>Done.</td>
</tr>
</tbody>
</table>

Wednesday, August 1, 12
Heapsort

• In the way that Heapsort was demonstrated in the previous slides, it could be implemented as follows:

```java
void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>();
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

• In the way that Heapsort was demonstrated in the previous slides, it could be implemented as follows:

```java
void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>();
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```

• This works fine, but does not operate in-place because the heap requires \(O(n)\) additional storage to the input array itself.
Heapsort

• In the way that Heapsort was demonstrated in the previous slides, it could be implemented as follows:

```java
void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>();
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```

```java
class HeapImpl12<T ...

    T[] _nodeArray;
    int _numNodes = 0;

    HeapImpl12<T> () {
        _nodeArray = (T[]) new Comparable[128];
    }
```

_nodeArray requires $O(n)$ additional storage to the user’s input array
Heapsort

• However, we can also implement Heapsort to work in-place.

• The “trick” is that the input array to Heapsort will serve as the HeapImpl12’s underlying storage.

• In HeapImpl12, instead of allocating a new _nodeArray, we will simply set it to the input array.
Heapsort

- We implement a new `HeapImpl12` constructor and use it in `heapsort`:

```java
void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```

```java
class HeapImpl12<T> ... {
    T[] _nodeArray;
    int _numNodes = 0;

    HeapImpl12<T> () {
        _nodeArray = (T[]) new Comparable[128];
    }
    HeapImpl12<T> (T[] array) {
        _nodeArray = array;
    }
}
```

Here we are offering a second constructor, in which the user can pass in an existing array as the heap's underlying storage. ==> No extra storage!
Heapsort

• Example -- let’s turn the following 8-element input array into a heap.

6 1 4 3 8 7 2 5

• The array above is the heap’s underlying storage (_nodeArray), and it is also the user’s input array.

• Initially, _numNodes = 0.

• Each time we “add” an element to the heap, _numNodes will increase by 1.
Heapsort

• Example -- let's turn the following 8-element input array into a heap.

```
6 1 4 3 8 7 2 5
```

Heapified elements      Non-heapified elements

```java
void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

• Example -- let's turn the following 8-element input array into a heap.

Heapified elements    Non-heapified elements
6 1 4 3 8 7 2 5

_heapNodes: 1

Increment _numNodes

void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
Heapsort

• Example -- let’s turn the following 8-element input array into a heap.

Heapified elements  Non-heapified elements

6 1 4 3 8 7 2 5

_call bubbleUp

void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
Heapsort

- Example -- let’s turn the following 8-element input array into a heap.

Heapified elements | Non-heapified elements
6 1 4 3 8 7 2 5

```java
void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

• Example -- let’s turn the following 8-element input array into a heap.

Heapified elements Non-heapified elements
6 1 4 3 8 7 2 5

_numNodes: 2

void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}

Call bubbleUp
Heapsort

• Example -- let's turn the following 8-element input array into a heap.

<table>
<thead>
<tr>
<th>Heapified elements</th>
<th>Non-heapified elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 1 4 3 8 7 2 5</td>
<td></td>
</tr>
</tbody>
</table>

```java
void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

- Example -- let’s turn the following 8-element input array into a heap.

6 1 4 3 8 7 2 5

_heapified elements_ _Non-heapified elements_

void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
Heapsort

- Example -- let’s turn the following 8-element input array into a heap.

   Heapified elements  Non-heapified elements
   6 1 4 3 8 7 2 5

   _numNodes: 4

   Increment _numNodes

```java
void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

Example -- let’s turn the following 8-element input array into a heap.

Heapified elements  Non-heapified elements
6 1 4 3 8 7 2 5

```java
void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

Example -- let’s turn the following 8-element input array into a heap.

6 3 4 1 8 7 2 5

_heapified elements   _non-heapified elements

void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
Heapsort

- Example -- let’s turn the following 8-element input array into a heap.

Heapified elements: 6 3 4 1 8 7 2 5
Non-heapified elements: 5

```
_numNodes: 5
Increment _numNodes
```

```java
void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

• Example -- let’s turn the following 8-element input array into a heap.

Heapified elements Non-heapified elements
6 3 4 1 8 7 2 5

```java
void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

- Example -- let's turn the following 8-element input array into a heap.

Heapified elements  Non-heapified elements
8  6  4  1  3  7  2  5

_keepNodes: 5

Call bubbleUp

Keep repeating this process...

```java
void heapsort (Integer[] array) {
    Heap12<Integer> heap = new HeapImpl12<Integer>(array);
    for (int i = 0; i < array.length; i++) {
        heap.add(array[i]);
    }
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

- Example -- let’s turn the following 8-element input array into a heap.

<table>
<thead>
<tr>
<th>Heapified elements</th>
<th>Non-heapified elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 6 7 5 3 4 2 1</td>
<td></td>
</tr>
</tbody>
</table>

_numNodes: 8

- We have now constructed a heap within the input array itself.

- This requires 0 extra storage.
Heapsort

• However, we’re still not done.

• We still have to call `removeLargest()` repeatedly, and store its result into the leftmost position of the sorted part of the array.

• Since we’re operating in-place, this will require that store the largest value `x` in a temporary variable.
Heapsort

- Given that the unsorted part of the array is now a valid heap, we can repeatedly call `removeLargest()` to populate the sorted part of the array:

```
Unsorted part       Sorted part
     8  6  7  5  3  4  2  1

_max:
_numNodes: 8
```

Save the heap’s largest element in `_max`, and then remove the largest element.

```java
void heapsort (Integer[] array) {
    ...
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

- Given that the unsorted part of the array is now a valid heap, we can repeatedly call removeLargest() to populate the sorted part of the array:

Unsorted part | Sorted part
---|---
8 6 7 5 3 4 2 1

_max: 8
_numNodes: 8

Save the heap’s largest element in _max, and then remove the largest element.

```java
void heapsort (Integer[] array) {
    ...
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

- Given that the *unsorted part* of the array is now a valid *heap*, we can repeatedly call `removeLargest()` to populate the *sorted part* of the array:

<table>
<thead>
<tr>
<th>Unsorted part</th>
<th>Sorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 6 7 5 3 4 2</td>
<td></td>
</tr>
</tbody>
</table>

```java
void heapsort (Integer[] array) {
    ... 
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```

Calling `removeLargest` requires us to `trickleDown` from the root.
Heapsort

- Given that the *unsorted part* of the array is now a valid *heap*, we can repeatedly call `removeLargest()` to populate the *sorted part* of the array:

    Unsorted part       Sorted part
    7  6  1  5  3  4  2

    _max: 8
    _numNodes: 7

    Calling `removeLargest` requires us to `trickleDown` from the root.

```java
void heapsort (Integer[] array) {
    ...
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

- Given that the unsorted part of the array is now a valid heap, we can repeatedly call `removeLargest()` to populate the sorted part of the array:

<table>
<thead>
<tr>
<th>Unsorted part</th>
<th>Sorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 4 5 3 1 2</td>
<td></td>
</tr>
</tbody>
</table>

```java
void heapsort (Integer[] array) {
    ...
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```

Calling `removeLargest()` requires us to `trickleDown` from the root.
Heapsort

- Given that the *unsorted part* of the array is now a valid *heap*, we can repeatedly call `removeLargest()` to populate the *sorted part* of the array:

  Unsorted part | Sorted part
  --------------|------------
  7 6 4 5 3 1 2 8

  `_max`:
  `_numNodes`: 7

Finally, we store `_max` into the *sorted part* of the array.

```java
void heapsort (Integer[] array) {
    ...
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

• Given that the *unsorted part* of the array is now a valid heap, we can repeatedly call `removeLargest()` to populate the *sorted part* of the array:

<table>
<thead>
<tr>
<th>Unsorted part</th>
<th>Sorted part</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 4 5 3 1 2</td>
<td>8</td>
</tr>
</tbody>
</table>

Save the heap’s largest element in `_max`, and then remove the largest element.

```java
void heapsort (Integer[] array) {
  ...
  for (int i = array.length - 1; array >= 0; array--) {
    array[i] = heap.removeLargest();
  }
}
```
Heapsort

• Given that the unsorted part of the array is now a valid heap, we can repeatedly call removeLargest() to populate the sorted part of the array:

Unsorted part: 2 6 4 5 3 1
Sorted part: 8

_max: 7
_numNodes: 7

Calling removeLargest requires us to trickleDown from the root.

```java
void heapsort (Integer[] array) {
    ...
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

- Given that the unsorted part of the array is now a valid heap, we can repeatedly call removeLargest() to populate the sorted part of the array:

Unsorted part          Sorted part
6 2 4 5 3 1            8
_max: 7
_numNodes: 7

Calling removeLargest requires us to trickleDown from the root.

```java
void heapsort (Integer[] array) {
    ...  
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

- Given that the unsorted part of the array is now a valid heap, we can repeatedly call `removeLargest()` to populate the sorted part of the array:

Unsorted part  Sorted part
6 5 4 2 3 1     8

_max: 7
_numNodes: 7

Calling `removeLargest` requires us to `trickleDown` from the root.

```java
void heapsort (Integer[] array) {
    ...
    for (int i = array.length - 1; array >= 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

- Given that the unsorted part of the array is now a valid heap, we can repeatedly call `removeLargest()` to populate the sorted part of the array:

```
6 5 4 2 3 1 7 8
```

Finally, we store `max` into the sorted part of the array.

```java
void heapsort (Integer[] array) {
    ...
    for (int i = array.length - 1; array > 0; array--) {
        array[i] = heap.removeLargest();
    }
}
```
Heapsort

• We repeatedly remove the largest element and store it into the leftmost slot of the unsorted part of the array, until the heap is empty.

• At that point, the array will be completely sorted.

• Since this required only one auxiliary variable (\(_\text{max}\)), the algorithm works in-place.
Heapsort

• In summary, heapsort is an in-place sorting algorithm whose best and worst case time costs are $O(n \log n)$.

• However, the algorithm is not stable because the heap ordering may cause the relative order of duplicate elements to become inverted.
Mergesort.
Approach 2: divide and conquer

• So far we’ve looked at sorting algorithms that partition the input array into a sorted part and unsorted part, and then “grow” the sorted part to be the entire array.

• An alternative approach altogether is based on the divide-and-conquer principle:

• To sort a list of size $n$:
  • Divide the list into two halves (approx. size $n/2$).
  • Sort each half independently using recursion.
  • Combine the 2 sorted lists of $n/2$ elements into 1 sorted list of $n$ elements.
Mergesort

• The first algorithm we examine that uses divide-and-conquer is **Mergesort**.

• Here’s the “main idea” behind the algorithm:
  
  • Suppose we have a *left list* and a *right list* that are *already sorted*.

  • To combine these two lists into one larger sorted list, we just:
    
    • Iterate through both lists simultaneously.

    • “Pick out” the smaller element from the current position of either the left or right list, and insert it into our *combined* list.
Merging two sorted lists

- Example:

<table>
<thead>
<tr>
<th>Left list</th>
<th>Right list</th>
<th>Combined list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 4 6</td>
<td>2 5 7 8</td>
<td></td>
</tr>
</tbody>
</table>

Iterate through both lists:

- Pick out the smaller element $x$ from the current position of either the left or right list;
- Advance the pointer of whichever list contained $x$;
- Then insert $x$ into the combined list.
Merging two sorted lists

- Example:

<table>
<thead>
<tr>
<th>Left list</th>
<th>Right list</th>
<th>Combined list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 4 6</td>
<td>2 5 7 8</td>
<td>1</td>
</tr>
</tbody>
</table>

Iterate through both lists:

*Pick out* the smaller element $x$ from the current position of either the left or right list;
*Advance* the pointer of whichever list contained $x$;
Then *insert* $x$ into the combined list.
Merging two sorted lists

- Example:

<table>
<thead>
<tr>
<th>Left list</th>
<th>Right list</th>
<th>Combined list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 4 6</td>
<td>2 5 7 8</td>
<td>1 2</td>
</tr>
</tbody>
</table>

Iterate through both lists:
- Pick out the smaller element $x$ from the current position of either the left or right list;
- Advance the pointer of whichever list contained $x$;
- Then insert $x$ into the combined list.
Merging two sorted lists

• Example:

<table>
<thead>
<tr>
<th>Left list</th>
<th>Right list</th>
<th>Combined list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 4 6</td>
<td>2 5 7 8</td>
<td>1 2 3</td>
</tr>
</tbody>
</table>

Iterate through both lists:

*Pick out* the smaller element $x$ from the current position of either the left or right list;

*Advance* the pointer of whichever list contained $x;$

*Then* *insert* $x$ into the combined list.
Merging two sorted lists

• Example:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 4 6</td>
<td>2 5 7 8</td>
<td>1 2 3 4</td>
</tr>
</tbody>
</table>

Iterate through both lists:
Pick out the smaller element \( x \) from the current position of either the left or right list;
Advance the pointer of whichever list contained \( x \);
Then insert \( x \) into the combined list.
Merging two sorted lists

- Example:

<table>
<thead>
<tr>
<th>Left list</th>
<th>Right list</th>
<th>Combined list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 4 6</td>
<td>2 5 7 8</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
</tr>
</tbody>
</table>

Iterate through both lists:

Pick out the smaller element \( x \) from the current position of either the left or right list;

Advance the pointer of whichever list contained \( x \);

Then insert \( x \) into the combined list.
Merging two sorted lists

- Example:

<table>
<thead>
<tr>
<th>Left list</th>
<th>Right list</th>
<th>Combined list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 4 6</td>
<td>2 5 7 8</td>
<td>1 2 3 4 5 6</td>
</tr>
</tbody>
</table>

Iterate through both lists:

- Pick out the smaller element $x$ from the current position of either the left or right list;
- Advance the pointer of whichever list contained $x$;
- Then insert $x$ into the combined list.
Merging two sorted lists

• Example:

<table>
<thead>
<tr>
<th>Left list</th>
<th>Right list</th>
<th>Combined list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 4 6</td>
<td>2 5 7 8</td>
<td>1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

Iterate through both lists:

Pick out the smaller element \( x \) from the current position of either the left or right list;

Advance the pointer of whichever list contained \( x \);

Then insert \( x \) into the combined list.
Merging two sorted lists

- Example:

<table>
<thead>
<tr>
<th>Left list</th>
<th>Right list</th>
<th>Combined list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 4 6</td>
<td>2 5 7 8</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

Done.

Iterate through both lists:

*Pick out* the smaller element $x$ from the current position of either the left or right list;

*Advance* the pointer of whichever list contained $x$;

Then *insert* $x$ into the combined list.
Mergesort

• Given a left list \((n/2\) elements) and a right list \((n/2\) elements), “merging” them into a combined list \((n\) elements) takes time \(O(n)\).

• However, it requires that we allocate a temporary array of size \(n\).

• Mergesort does not operate in-place.

• After merging, we copy the elements in the temporary array back into the input array.

* Except when using a linked-list representation.
Mergesort

• Given a procedure to *merge two sorted lists*, we can define a *recursive sorting algorithm* in the following way:

  • Given an input array:
    
    • If its length is 1, then it’s already sorted.
    
    • Else:
      
      • Divide the list into two halves.
      
      • Recursively sort each half.
      
      • Merge their results into one combined list.
Mergesort

• Mergesort’s pseudocode:

```java
void mergesort (array) {
    If array.length == 1, then do nothing.           Base case
    Else:
        Split array evenly into leftArray and rightArray.
        mergesort(leftArray);
        mergesort(rightArray);
        Merge the leftArray and rightArray into array
}
```

• Let’s see how it works in practice...
Example: First stage: recursively divide until we reach the base case.

```
void mergesort (array) {
  if array.length == 1, then do nothing.
  else:
    Split array evenly into leftArray and rightArray.
    mergesort(leftArray);
    mergesort(rightArray);
    Merge the leftArray and rightArray into array
}
```
Mergesort

• Example: First stage: recursively divide until we reach the base case.

6 1 4 3 8 7 2 5

6 1 4 3
8 7 2 5

void mergesort (array) {
    If array.length == 1, then do nothing.
    Else:
        Split array evenly into leftArray and rightArray.
        mergesort(leftArray);
        mergesort(rightArray);
        Merge the leftArray and rightArray into array
}
Mergesort

- Example: First stage: recursively divide until we reach the base case.

\[ 6 \ 1 \ 4 \ 3 \ 8 \ 7 \ 2 \ 5 \]

\[ 6 \ 1 \ 4 \ 3 \ \ 8 \ 7 \ 2 \ 5 \]

\[ 6 \ 1 \ 4 \ 3 \ \ 8 \ 7 \ 2 \ 5 \]

\[ 6 \ 1 \ 4 \ 3 \ \ 8 \ 7 \ 2 \ 5 \]

\[ 6 \ 1 \ 4 \ 3 \ \ 8 \ 7 \ 2 \ 5 \]

\[ 6 \ 1 \ 4 \ 3 \ \ 8 \ 7 \ 2 \ 5 \]

\[ 6 \ 1 \ 4 \ 3 \ \ 8 \ 7 \ 2 \ 5 \]

\[ 6 \ 1 \ 4 \ 3 \ \ 8 \ 7 \ 2 \ 5 \]

void mergesort (array) {
    If array.length == 1, then do nothing.
    Else:
        Split array evenly into leftArray and rightArray.
        mergesort(leftArray);
        mergesort(rightArray);
        Merge the leftArray and rightArray into array
}
Mergesort

- Example:

Each of these is a “list” (size 1) passed to a recursive call to Mergesort.

6 1 4 3 8 7 2 5

void mergesort (array) {
    If array.length == 1, then do nothing.
    Else:
        Split array evenly into leftArray and rightArray.
        mergesort(leftArray);
        mergesort(rightArray);
        Merge the leftArray and rightArray into array
}
Mergesort

- Example: Second stage: merge each pair of sorted sub-lists.

```plaintext
void mergesort (array) {
    If array.length == 1, then do nothing.
    Else:
        Split array evenly into leftArray and rightArray.
        mergesort(leftArray);
        mergesort(rightArray);
        Merge the leftArray and rightArray into array
}
```
Mergesort

- Example:

  Second stage: merge each pair of sorted sub-lists.

  1 3 4 6  2 5 7 8

  1 6  3 4  7 8  2 5

  6 1  4 3  8 7  2 5

  Merge the two sub-lists.

void mergesort (array) {
    If array.length == 1, then do nothing.
    Else:
        Split array evenly into leftArray and rightArray.
        mergesort(leftArray);
        mergesort(rightArray);
        Merge the leftArray and rightArray into array
}
Mergesort

- Example: Second stage: merge each pair of sorted sub-lists.

1 2 3 4 5 6 7 8

Merge the two sub-lists.

1 3 4 6
2 5 7 8

1 6 3 4 7 8 2 5

6 1 4 3 8 7 2 5

```c
void mergesort (array) {
    If array.length == 1, then do nothing.
    Else:
        Split array evenly into leftArray and rightArray.
        mergesort(leftArray);
        mergesort(rightArray);
    Merge the leftArray and rightArray into array
}
```
Mergesort

• Example:

1 2 3 4 5 6 7 8

1 3 4 6 2 5 7 8

1 6 3 4 7 8 2 5

6 1 4 3 8 7 2 5

void mergesort (array) {
    If array.length == 1, then do nothing.
    Else:
        Split array evenly into leftArray and rightArray.
        mergesort(leftArray);
        mergesort(rightArray);
        Merge the leftArray and rightArray into array
}
Mergesort

- Example:

  1 2 3 4 5 6 7 8
  1 3 4 6          2 5 7 8
  1 6     3 4     7 8     2 5
  6   1   4   3   8   7   2   5

- The depth of this recursive call stack is the number of times we can divide n by 2, i.e., $O(\log n)$. 
Mergesort

- Example:

```
1 2 3 4 5 6 7 8
1 3 4 6         2 5 7 8
1 6     3 4     7 8     2 5
6   1   4   3   8   7   2   5
```

- At each level, each element in the input array had to be “touched” once (for the merge operation).

- In total: $O(\log n) \times n = O(n \log n)$. 
• Because Mergesort’s dividing and merging requires
  the same number of operations \textit{regardless} of the
  particular input, Mergesort’s \textit{best case} and \textit{worst
  case} time complexities are both $O(n \log n)$.

• Mergesort is \textit{stable} as long as the \textit{merge}
  procedure selects the left array’s $x$ in the case of ties.