CS062
DATA STRUCTURES AND ADVANCED PROGRAMMING

12: Insertion Sort & Mergesort

Alexandra Papoutsaki
she/her/hers

Tom Yeh
he/him/his
Lecture 12: Insertion Sort & Mergesort

- Insertion Sort
- Mergesort
Insertion sort

Keep a *partially sorted subarray* on the left and an *unsorted subarray* on the right.

Repeat:

- Examine the next element in the unsorted subarray.
- Insert this element by exchanging with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Insert this element by exchanging with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Insertion sort

3 44 38 5 47 1 36 26

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Exchange this element with every entry to the left that is greater.
  - Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
INSERTION SORT

Insertion sort

| 3  | 44 | 38 | 5  | 47 | 1  | 36 | 26 |

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Exchange this element with every entry to the left that is greater.
  - Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Exchange this element with every entry to the left that is greater.
  - Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Insertion sort

```
3  38  5  44  47  1  36  26
```

Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Insertion sort

Repeated steps:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
INSERTION SORT

Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Insertion sort

- **Repeat:**
  - Examine the next element in the unsorted subarray.
  - Exchange this element with every entry to the left that is greater.
  - Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeated elements:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.

Insertion sort

3 5 1 38 44 47 36 26
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
INSERTION SORT

Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
**Insertion sort**

Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
INSERTION SORT

Insertion sort

- Repeat:
  - Examine the next element in the unsorted subarray.
  - Exchange this element with every entry to the left that is greater.
  - Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.

Insertion sort

```
1 3 5 36 38 44 47 26
```
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
INSERTION SORT

Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Insertion sort

Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
Repeat:

- Examine the next element in the unsorted subarray.
- Exchange this element with every entry to the left that is greater.
- Move subarray boundaries one element to the right.
2.1 Insertion Sort Demo

Demo with Cards
In case you didn’t get this...

- [https://www.youtube.com/watch?v=ROalU379l3U](https://www.youtube.com/watch?v=ROalU379l3U)
Insertion sort

```java
public static void sort(Comparable[] a) {
    // for loop to iterate through each element of the array

    // Moving right to left, exchange a[i] with every larger
    // entry to its left

}
```
Insertion sort

```java
public static void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        for (int j = i; j > 0; j--) {
            if (less(a[j], a[j-1])) {
                exch(a, j, j-1);
            } else {
                break;
            }
        }
    }
}
```

> **Invariants:** At the end of each iteration $i$:

- The array $a$ is sorted in ascending order for the first $i+1$ elements $a[0...i]$
**Insertion sort: mathematical analysis for worst-case**

```java
public static void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        for (int j = i; j > 0; j--) {
            if (less(a[j], a[j-1])) {
                exch(a, j, j-1);
            } else {
                break;
            }
        }
    }
}
```

- Comparisons:
- Exchanges: ?
- In-place?
- Stable?
Insertion sort: mathematical analysis for \textbf{worst-case}

```
public static void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        for (int j = i; j > 0; j--) {
            if (less(a[j], a[j-1])) {
                exch(a, j, j-1);
            } else {
                break;
            }
        }
    }
}
```

\begin{itemize}
\item \textbf{Comparisons:} $0 + 1 + 2 + \ldots + (n - 2) + (n - 1)\sim n^2/2$, that is $O(n^2)$.
\item \textbf{Exchanges:} ?
\item \textbf{In-place?}
\item \textbf{Stable?}
\end{itemize}
**INSERTION SORT**

**Insertion sort: mathematical analysis for worst-case**

```java
public static void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        for (int j = i; j > 0; j--) {
            if (less(a[j], a[j-1])) {
                exch(a, j, j-1);
            } else {
                break;
            }
        }
    }
}
```

- **Comparisons**: $0 + 1 + 2 + \ldots + (n - 2) + (n - 1) \sim n^2 / 2$, that is $O(n^2)$.

- **Exchanges**: $0 + 1 + 2 + \ldots + (n - 2) + (n - 1) \sim n^2 / 2$, that is $O(n^2)$.

- Worst-case running time is **quadratic**. Worst case = array sorted in reverse order.

- Every element moves all the way to the left.

- **In-place**, requires almost no additional memory.

- **Stable**
Insertion sort: average and best case

public static void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        for (int j = i; j > 0; j--) {
            if (less(a[j], a[j-1])) {
                exch(a, j, j-1);
            } else {
                break;
            }
        }
    }
}

- **Average case**: quadratic for both comparisons and exchanges $\sim n^2/4$ when sorting a randomly ordered array. (2X faster than selection sort on average)
  - Expect each entry to move halfway back: $0 + 0.5 + 1 + ...+(n-1)/2 \sim (n/2)*(n/2) \sim n^2/4$

- **Best case**: $n - 1$ comparisons (validate) and 0 exchanges for an already sorted array.

https://www.toptal.com/developers/sorting-algorithms/insertion-sort
Practice Time (Use your cards)

- Using insertion sort, sort the array with elements [12,10,16,11,9,7].
- Visualize your work for every iteration of the algorithm.
## Insertion Sort

### Answer

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>12</td>
<td>10</td>
<td>16</td>
<td>11</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>2nd</td>
<td>10</td>
<td>12</td>
<td>16</td>
<td>11</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>3rd</td>
<td>10</td>
<td>12</td>
<td>16</td>
<td>11</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>4th</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>5th</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>last</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
Insertion Sort

- For partially-sorted arrays, insertion sort runs in *linear time*
- Number of exchanges equals number of *inversions*
- *Inversion* = pair of keys that are out of order

Partially sorted examples
- 1) Appending a subarray of size 10 to a sorted subarray of size N
- 2) An array of size N with only 10 entries out of place
Lecture 12: Insertion Sort & Mergesort

- Insertion sort
- Comparators
- Mergesort
Comparable

- Interface with a single method that we need to implement:
  \[\text{public int compareTo(T that)}\]

- Implement it so that \(v\).compareTo(\(w\)):
  - Returns \(>0\) if \(v\) is greater than \(w\).
  - Returns \(<0\) if \(v\) is smaller than \(w\).
  - Returns \(0\) if \(v\) is equal to \(w\).

- Corresponds to natural ordering.
How to make your class T comparable?

1. Implement Comparable<T> interface.

2. Implement compareTo(T that) method to compare this T object to that based on natural ordering.
Comparator

- Sometimes the natural ordering is not the type of ordering we want.
- Comparator is an interface which allows us to dictate what kind of ordering we want by implementing the method:
  ```java
  public int compare(T this, T that)
  ```
- Implement it so that `compare(v, w)`:  
  - Returns >0 if v is greater than w.
  - Returns <0 if v is smaller than w.
  - Returns 0 if v is equal to w.
How to define an alternative ordering for your class T?

1. Make a new class that implements Comparator<T> interface.

2. Implement compare(T t1, T t2) method to compare t1 object to t2 based on an alternative ordering.

3. Alternatively, implement an anonymous inner class:

```java
public static Comparator<T> nameOfComparator = new Comparator<T>(){
    @Override // indicates method overriding the superclass' method
    public int compare(T t1, T t2) {
        //return something;
    }
};
```
The Java Collections Framework

Collections

- Set
  - SortedSet
  - NavigableSet
- List
  - AbstractSet
  - LinkedList
- Queue
  - Deque
  - AbstractQueue
- Collection
  - Iterable
  - AbstractList
  - AbstractSequentialList
  - AbstractCollection

Alternative sorting of Collections

- Collections class contains:
  - static <T> void sort(List<T> list, Comparator<? super T> c)
  - Collections.sort(list, someComparator);
  - Collections.sort(list, new ExternalComparatorClass()); or:
  - Collections.sort(list, T.InnerAnonymousClass);
- If list’s elements do not implement Comparable or cannot be compared with Comparator, throw ClassCastException.
Example: Natural and alternative sorting for Employees


https://stackoverflow.com/questions/2266827/when-to-use-comparable-and-comparator
Lecture 12: Insertion Sort & Mergesort

- Insertion sort
- Comparators
- Mergesort
Lecture 12: Insertion Sort & Mergesort

- Mergesort
Basics

- Invented by John von Neumann in 1945
- Algorithm sketch:
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge the two halves

Merging two already sorted halves into one sorted array

Copy to auxiliary array
Merging Example - copying to auxiliary array

Array a

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Merging Example - copying to auxiliary array

Array a

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Array aux

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Mergesort

Merging Example - copy elements back to original array in order

Maintain 3 indices: i, j, k

Array a

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

i     j
Merging Example - copy elements back to original array in order

Compare minimum in each subarray

Array a (sorted result)

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

i j k
Merging Example - copy elements back to original array in order

Copy smaller element back to a, increment indices i and j

Array a (sorted result)

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

k

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

i j
**Mergesort**

Merging Example - copy elements back to original array in order

Compare

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

\(k\)

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

\(i\) \(j\)
Mergesort

Merging Example - copy elements back to original array in order

Compare

Array a (sorted result)

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

k

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

i    j
Merging Example - copy elements back to original array in order

Compare

Array a (sorted result)

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

k

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

i   j
Merging Example - copy elements back to original array in order

Compare

Array a (sorted result)

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>H</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

k

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

i          j
Mergesort Example - copy elements back to original array in order

Compare

Array a (sorted result)

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>H</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

k

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

i  j
Merging Example - copy elements back to original array in order

Compare

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Array a (sorted result)

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Merging Example - copy elements back to original array in order

Compare

Array a (sorted result)

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>L</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

k

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

i, j
Merging Example - copy elements back to original array in order

Compare

Array a (sorted result)

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>L</th>
<th>M</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

i

j
Merging Example - copy elements back to original array in order

Compare

Array a (sorted result)

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>L</th>
<th>M</th>
<th>O</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

i j k
Merging Example - copy elements back to original array in order

Compare

Array a (sorted result)

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>L</th>
<th>M</th>
<th>O</th>
<th>R</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

i    j
Merging Example - copy elements back to original array in order

Compare

Array a (sorted result)

```
AGHILMORS
0123456789
```

Array aux

```
AGLORHMIST
0123456789
```

i  j  k
Merging Example - copy elements back to original array in order

Array aux

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>L</th>
<th>M</th>
<th>O</th>
<th>R</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Array a (sorted result)

<table>
<thead>
<tr>
<th>A</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>L</th>
<th>M</th>
<th>O</th>
<th>R</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

k

i

j
Merging two already sorted halves into one sorted array

private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    for (int k = lo; k <= hi; k++)  // copy to aux array
        aux[k] = a[k];

    int i = lo, j = mid + 1;       // lo and mid+1 are the start of the 2 sorted halves

    for (int k = lo; k <= hi; k++) {
        if (i > mid)  // ran out of elements in the left subarray
            a[k] = aux[j++];
        else if (j > hi)  // ran out of elements in the right subarray
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))  // Compares left and right subarray
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
2.2 Merging Demo
Practice time

How many calls does `merge()` make to `less()` in order to merge two already sorted subarrays, each of length \( n/2 \) into a sorted array of length \( n \)?

A. \( \sim \frac{1}{4}n \) to \( \frac{1}{2}n \)
B. \( \frac{1}{2}n \)
C. \( \sim \frac{1}{2}n \) to \( n \)
D. \( \sim n \)

```java
private static void merge(Comparable[] a, Comparable[] aux, int lo, int hi) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) //ran out of elements in the left subarray
            a[k] = aux[j++];
        else if (j > hi) //ran out of elements in the right subarray
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
}
```
Answer

How many calls does `merge()` make to `less()` in order to merge two already sorted subarrays, each of length \( n/2 \) into a sorted array of length \( n \)?

C. \( \sim \frac{1}{2}n \) to \( n \), that is at most \( n - 1 \) or \( O(n) \)

**Best case example**
Merging \([1,2,3] \) and \([4,5,6] \) requires 3 calls to `less()` (1 with 4, 2 with 4, 3 with 4).

**Worst case example**
Merging \([1,3,5] \) and \([2,4,6] \) requires 5 calls to `less()` (1 with 2, 2 with 3, 3 with 4, 4 with 5, 5 with 6)
Mergesort - the quintessential example of divide-and-conquer

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;  // Computes midpoint
    sort(a, aux, lo, mid);  // Sort the first half
    sort(a, aux, mid+1, hi);  // Sort the second half
    merge(a, aux, lo, mid, hi);  // Merge the 2 halves
}

public static void sort(Comparable[] a) {
    Comparable[] aux = new Comparable[a.length];  // Create aux array
    sort(a, aux, 0, a.length - 1);  // Recursively call sort
}
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

public static void sort(Comparable[] a) {
    Comparable[] aux = new Comparable[a.length];
    sort(a, aux, 0, a.length - 1);
}

sort([M, E, R, G, E, S, R, T]) calls
sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null, null], 0, 7) where the array of nulls is the auxiliary array, lo = 0 and hi = 7.
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null, null], 0, 7) calculates the mid = 3 and calls recursively sort on the left subarray, that is sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null, null], 0, 3), where lo = 0, hi = 3
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null, null], 0, 3) calculates the mid = 1 and calls recursively sort on the left subarray, that is sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null, null], 0, 1), where lo = 0, hi = 1
sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null, null], 0, 1) calculates the mid = 0 and calls recursively sort on the left subarray, that is sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null, null], 0, 0), where lo = 0, hi = 0.
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) {
        return;
    }
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}

sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null, null], 0, 0) finds hi <= lo and returns.
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) {
        return;
    }
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null, null], 1, 1) finds hi <= lo and returns.
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

sort([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null, null], 0, 1) merges the two subarrays that is calls merge([M, E, R, G, E, S, R, T], [null, null, null, null, null, null, null, null, null], 0, 0, 1), where lo = 0, mid = 0, and hi = 1. The resulting partially sorted array is [E, M, R, G, E, S, R T].
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) {
        return;
    }
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null, null], 0, 3) calls recursively sort on the right subarray, that is sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null, null], 2, 3), where lo = 2, hi = 3
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null, null], 2, 3) calculates the mid = 2 and calls recursively sort on the left subarray, that is sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null, null], 2, 2), where lo = 2, hi = 2
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) {
        return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
    }
}

sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null, null], 2, 2) finds hi <= lo and returns.
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null, null], 2, 3) calls recursively sort on the right subarray, that is sort([E, M, R, G, E, S, R, T], [M, E, null, null, null, null, null, null, null], 3, 3), where lo = 3, hi = 3
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

The code block illustrates a recursive sorting algorithm using a divide and conquer strategy. The `sort` method recursively divides the array into subarrays until single-element subarrays are reached, then merges them back together in a sorted fashion. The `merge` method is used to combine two sorted subarrays into one sorted array. The given example demonstrates how the `sort` method works with a specific array and indices, resulting in a partially sorted array.

### Java Code

```java
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) {
        return;
    }
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}
```

### Example

The `sort` method is called with the array `[E, M, R, G, E, S, R, T]` and subarrays starting from indices `lo = 2` and `hi = 3`. The resulting partially sorted array is `[E, M, G, R, E, S, R, T]`. This shows how the algorithm works by recursively sorting subarrays and merging them back together, ensuring the entire array is sorted.

### Diagram

The diagram visualizes the recursive calls of the `sort` method, breaking down the array into smaller subarrays, sorting them, and then merging them back together.
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) {
        return;
    }
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) {
        return;
    }
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

the mid = 5 and calls recursively sort on the left subarray, that is sort([E, G, M, R, E, S, R, T],
[E, M, G, R, null, null, null, null], 4, 5), where lo = 4, hi = 5.
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) {
        return;
    }
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}


private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```java
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}


private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
**private static void** sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}

Practice time

Which of the following subarray lengths will occur when running mergesort on an array of length 10?

A. { 1, 2, 3, 5, 10 }
B. { 2, 4, 6, 8, 10 }
C. { 1, 2, 5, 10 }
D. { 1, 2, 3, 4, 5, 10 }
Answer

Which of the following subarray lengths will occur when running mergesort on an array of length 10?

A. \{ 1, 2, 3, 5, 10 \}
Good algorithms are better than supercomputers

- Your laptop executes $10^8$ comparisons per second
- A supercomputer executes $10^{12}$ comparisons per second

<table>
<thead>
<tr>
<th>Computer</th>
<th>Insertion sort</th>
<th>Mergesort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Thousand inputs</td>
<td>Million inputs</td>
</tr>
<tr>
<td><strong>Home</strong></td>
<td>Instant</td>
<td>2 hours</td>
</tr>
<tr>
<td><strong>Supercomputer</strong></td>
<td>Instant</td>
<td>1 second</td>
</tr>
</tbody>
</table>
Analysis of comparisons

- We will assume that that \( n \) is a power of 2 \((n = 2^k, \text{ where } k = \log_2 n)\) and the number of comparisons \( T(n) \) to sort an array of length \( n \) with merge sort satisfies the recurrence:
  - \( T(n) = T(n/2) + T(n/2) + (n - 1) = O(n \log n) \)

- Number of array accesses (rather than exchanges, here) is also \( O(n \log n) \).
Mergesort uses \( \leq n \log n \) compares to sort an array of length \( n \)

If \( n = 4 \), 2 levels

If \( n = 8 \), 3 levels

If \( n = 16 \), 4 levels

... 

If \( n = 2^k \), \( k \) levels,

or \( k = \log_2 n \)

\[
O\left( \sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i} \right) = O\left( \sum_{i=0}^{k} n \right) = O(k \cdot n) \iff O(n \cdot \log n)
\]

\((\log n \text{ levels}) \times (n \text{ comparisons}) = O(n \log n)\)
Any algorithm with the same structure takes $n \log n$ time

```java
public static void f(int n) {
    if (n == 0)
        return;
    f(n/2);
    f(n/2);
    linear(n);
}
```
Mergesort basics

- Auxiliary memory is proportional to $n$, as $\text{aux}[]$ needs to be of length $n$ for the last merge.

- At its simplest form, merge sort is not an in-place algorithm.

- There are modifications for halting the size of the auxiliary array but in-place merge is very hard.

- **Stable**: Look into $\text{merge()}$, if equal keys, it takes them from the left subarray.
  - So is insertion sort, but not selection sort.
Practical improvements for Mergesort

- Use insertion sort for small subarrays.
- Stop if already sorted.
- Eliminate the copy to the auxiliary array by saving time (not space).

```java
private static void sort(Comparable[] src, Comparable[] dst, int lo, int hi) {
    if (hi <= lo + 7) {
        insertionSort(dst, lo, hi);
        return;
    }

    int mid = lo + (hi - lo) / 2;
    sort(dst, src, lo, mid);
    sort(dst, src, mid+1, hi);

    if (!less(src[mid+1], src[mid])) {
        for (int i = lo; i <= hi; i++) dst[i] = src[i];
        return;
    }

    merge(src, dst, lo, mid, hi);
}

For years, Java used this version to sort Collections of objects.
## Sorting: the story so far

<table>
<thead>
<tr>
<th></th>
<th>In place</th>
<th>Stable</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>X</td>
<td></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$n$ exchanges</td>
</tr>
<tr>
<td>Insertion</td>
<td>X</td>
<td>X</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>Use for small arrays or partially</td>
</tr>
<tr>
<td>Merge sort</td>
<td>X</td>
<td></td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>Guaranteed performance; stable</td>
</tr>
</tbody>
</table>
Lecture 13: Mergesort

- Mergesort
Readings:

- **Textbook:**
  - Chapter 2.2 (pages 270-277)

- **Website:**
  - Mergesort: [https://algs4.cs.princeton.edu/22mergesort/](https://algs4.cs.princeton.edu/22mergesort/)
  - Code: [https://algs4.cs.princeton.edu/22mergesort/Merge.java.html](https://algs4.cs.princeton.edu/22mergesort/Merge.java.html)

Practice Problems:

- 2.2.1-2.2.2, 2.2.11
Readings:

- **Textbook:**
  - Chapter 2.1 (pages 244-262), Chapter 2.1 (Page 247), Chapter 2.5 (Pages 338-339)

- **Website:**
  - Elementary sorts: [https://algs4.cs.princeton.edu/21elementary/](https://algs4.cs.princeton.edu/21elementary/)

- **Oracle documentation:**
  - Collections: [https://docs.oracle.com/javase/tutorial/collections/intro/index.html](https://docs.oracle.com/javase/tutorial/collections/intro/index.html)
  - Comparable: [https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)
  - Comparator: [https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html)

Practice Problems:

- 2.1.1-2.1.8