Comparable, lists and Sorting Algorithms

COSC 1P03 – Lecture 10 (Spring 2024)

Maysara Al Jumaily

amaysara@brocku.ca

Brock University

Tuesday July 02, 2024

Total slides: 33

Lecture Outline

The Comparable Interface I

- Suppose we have an unsorted array of `int`egers and wanted to sort it, how can that happen?
- In Java, we have a mechanism from the standard library for this type of scenario through `import java.util.Arrays;`.
- Here is a minimal example:

```
int[] data = new int[]{1, 6, 3, 7, 2};//unsorted array
Arrays.sort(data);//the array will be sorted after this
//Printing the array after it has been sorted.
for (int value : data) { //A for loop from i=0 to length is valid too
  System.out.print(value + "\t");
}
```

- The interesting thought is that Java knows how to sort `String`s and primitive types (`int`, `long`, `short`, `char`, `byte`, `float` and `double`, except `boolean` because it is just `true`/`false`).
- The question is how can we make a custom object and have Java sort it based on our definition? Well, we would have to use the `Comparable` interface.
- The `String` class and the wrapper classes of the mentioned primitive type above (*i.e.*, `Integer`, `Long`, etc.) all `implements` the `Comparable` interface.

The Comparable Interface II

- The `Comparable` interface is parameterized and has one and only one method to implement: the `compareTo` method.
- Since we want compare two objects, it would make sense to have both objects of the same type. For example, if our class is `Student`, then we *should* (not a must but should) to compare it to another `Student`. Logically speaking, it doesn't make sense to compare it with a `Turtle`, or `ASCIIDataFile`, etc. We need to compare it to itself, `Student`. Coding-wise, we are allowed to compare it to a different object.
- Our `Student` class should `implements` the `Comparable<Student>` interface.
- The method `compareTo` accepts a generic type (or `Student` in our case as we substituted `Student` in the generics `< ... >`). We will use a key to sort by (such as `studentID`, or `age` or `grade`, or `major`, etc.). It will return an `int`eger that could either be `-1`, `0` or `1`:
  - `-1`: Current object's key is **smaller** than the key of the parameter passed
  - `0`: Current object's key is **equal** than the key of the parameter passed
  - `1`: Current object's key is **larger** than the key of the parameter passed
- To reverse the order (*i.e.*, ascending to descending or vice-versa), instead of returning `-1`, return `1`, and vice-versa.
- The type of the `compareTo` **MUST** match the type passed in the generics (*i.e.*, `< ... >`).

The Comparable Interface III

- Let us assume that our Student object contains three instance variables: `id`, `name`, and `grade`, along with just getter methods (no setters for this minimal example but would work regardless):

```java
public class Student implements Comparable<Student> {
  private String name;
  private String id;
  private double grade;
  public Student(String id, String name, double grade) {
    this.name = name;
    this.id = id;
    this.grade = grade;
  }
  public String getName() { return name; }
  public String getId() { return id; }
  public double getGrade() { return grade; }
  public String toString() { return grade+", "+id+", "+name;}
  public int compareTo(Student studentPassed) {
    // Discussed next slide ...
  }
}
```

The Comparable Interface IV

- We are currently inside of the `Student` class, and the three instance variables are what we will refer to as "current student". The parameter passed will be the "other student".
- We use any of the `name`, `grade` or `id` as the key for comparison. We will stick to one and only one key. Since Java supports mathematical comparison operators (*i.e.*, `<`, `<=`, `==`, `!=`, `>`, `>=`), we can use the grade because it is a `double`:

```java
public int compareTo(Student studentPassed) {
  if(grade < studentPassed.getGrade()){
    //current student less than other student
    return -1;
  } else if (grade == studentPassed.getGrade()){
    //both students are equal
    return 0;
  } else {
    //current student is larger than the other student
    return 1;
  }
}
```

- The sort will be ascending (smallest first to largest last).

The Comparable Interface V

- Let us create a `Test` class, declare and initialize an array of `Student`s and apply the sorting logic we created from the previous slide:

```java
Student[] students = new Student[]{
  new Student("111111", "John Smith", 74),
  new Student("444444", "Mikey Glass", 92),
  new Student("333333", "Alex Stevenson", 88),
  new Student("777777", "Kelvin Chanel", 51),
  new Student("555555", "Dwight K. Schrute", 66),
  new Student("888888", "Connor Joey", 70),
  new Student("222222", "Matteo Richards", 71),
  new Student("999999", "Cranjis McBasketball", 31),
  new Student("666666", "Ronald Aaron", 42),
};

Arrays.sort(students); //need to use the sort method given by Java
for (Student s : students) {
  System.out.println(s);
}
```

- Ensure to add the import statement at the top: `import java.util.Arrays;`.
- The output is found on the next slide.

The Comparable Interface V

- The output when the array of `Student`s is sort based on the `grade` key is:

```
31.0, 999999, Cranjis McBasketball
42.0, 666666, Ronald Aaron
51.0, 777777, Kelvin Chanel
66.0, 555555, Dwight K. Schrute
70.0, 888888, Connor Joey
71.0, 222222, Matteo Richards
74.0, 111111, John Smith
88.0, 333333, Alex Stevenson
92.0, 444444, Mikey Glass
```

- Note that the `Student` class overrides the `toString` method, hence the student data is printed accordingly instead of the typical gibberish, such as `Student@1a3cc4d`.
- In conclusion: To allow Java to sort an array of customly created object, ensure the class `implements` the parameterized `Comparable` and pass the same type in `< ... >`. The `compareTo` method must accept one parameter that matches same type specified in the generics and will either return `-1`, `0` or `1`.

Lists I

- A *list* is a cursored data structure that allows the operations of insert/removing be done relative to the cursor, not just from the front/rear.
- It can be implemented as a linked list or based on an array (variable-size array). We will implement the array approach.
- The start of the list will be at index `0` and the rear will be at index `array.length - 1`, or the rear is up to the size, as it is a variable-sized array.
- The cursor is a variable that stores the index value of the cell.
- The `cursor` is an `int` that starts at index `0` and has the ability to move to the next node(s) at indices `1`, `2`, etc. It could be at the front, middle, or rear. Regardless of where it is, the operation of insertion/removing will be performed relative to it.
- For example, suppose that we have cursor at the front of the elements and wanted to:
  - **Insert**: We will shift all of the elements in the array one cell to the right, then insert at index `0`.
  - **Remove**: We will shift all of the elements one cell to the left to override the value at index `0`.
  - In general, both operations will take $\mathcal{O}(n)$.

Lists III – The Interface

- The interface of `List` contains the following operations:

```
public interface List<E extends Comparable<E>> extends Iterable<E>{
  public void add(E item); //insert a value at the index of cursor
  public E remove(); //removes the value cursor is pointing on

  public E get(); //returns the value cursor is pointing to

  public boolean empty(); //if the list is empty or not

  public int length(); //number of elements in variable-sized array

  public void toFront(); // moves cursor to front

  public void advance();//moves cursor to the next value

  public boolean offEnd(); //cursor not pointing to valid cell

  public void find(E element); //If value exists in list starting
                               //from cursor to end of array, make
                               //cursor point to it, else, offEnd
}
```

Lists II – The Interface

- The interface of `List` is parameterized as `<E extends Comparable<E>>` and also `extends Iterable<E>`.
- The `<E extends Comparable<E>>` part is referring to some type `E`. We are **not** allowed to pass any type here, instead, our type **MUST** `implements` the `Comparable<E>` interface (*i.e.*, it compares the same type as itself, similar to the `Student` class that compared another `Student` type).
- In other words, the type that we will pass to our `List` must implements the `Comparable<...>` interface and compare itself.
- The `extends Iterable<E>` part guarantees that the object we will pass as `E` `implements` the `Iterable` interface (that loops through the same type of elements, `E`), to ensure it could be used in a while-has-next logic and `for`-each loop.

Sorting Algorithms I

- Sorting algorithms are algorithms that places elements in an array in some order, ascending or descending order.
- There are two concepts to understand about sorting algorithms:
  - **In-situ (or in-place)**: To sort n elements, an in-situ algorithm will use $\mathcal{O}(C)$ of extra space (*i.e.*, one temporary variable, or two, or three, or *constant* number of extra variables).
  - **Stable**: A stable algorithm is an algorithm that will leave equal element in the same order as they appeared. For instance, the next example has two 5's, where the red 5 is on the left and blue 5 on the right. A stable will ensure the order of equal elements kept the same:

3	5	4	1	5	2	Unsorted Array
1	2	3	4	5	5	Stable Sorted Array
1	2	3	4	5	5	Unstable Sorted Array

- There will be four sorting algorithms to cover. Note that the algorithms' properties (such as in-situ or stable) could change depending on the implementation. The implementation code has been provided for each algorithm.

Sorting Algorithms II

- The table below along with their worst time complexity (and examples) and whether each is in-situ or stable:

Algorithm	**Complexity**	**In-situ (constant space)**	**Stable (equal keys same order)**	**Time to sort 500,000 elements (milliseconds)**
Bubble Exchange sort	$\mathcal{O}\left(n^2\right)$	Yes $\mathcal{O}(C)$	Yes	323,320
Selection sort	$\mathcal{O}\left(n^2\right)$	Yes $\mathcal{O}(C)$	No	77,895
Insertion sort	$\mathcal{O}\left(n^2\right)$	Yes $\mathcal{O}(C)$	Yes	12,541
Merge sort	$\mathcal{O}(n \log n)$	No $\mathcal{O}(n)$	Yes	33

- Yes, merge sort is about 10,000 faster than bubble/exchange sort!
- An honourable mention is quicksort, which has a time complexity of $\mathcal{O}(n \log n)$. It is less fast than merge sort, but doesn't use $\mathcal{O}(n)$ of extra space, just constant. Most programming languages use quicksort when sorting because it is quick and requires a constant amount of storage.

Bubble/Exchange Sort I

- Bubble sort will take $\mathcal{O}\left(n^2\right)$ operations to complete the sort.
- Let us sort in ascending order (least to greatest).
- The first iteration is going over all of the n elements. Compare the first two values, ensure the smaller value on the left and larger value on the right. Either swap them if not in order, or don't do anything and continue onto the next step.
- Compare the larger value from previous step to the value next to it.
- Swap if they are not in order, otherwise, continue to the next step.
- Eventually, we will have shift the largest value to the far-right.
- Now, we have one element that is sorted, and $n-1$ elements to sort, the first iteration is done.
- The second iteration is to start at the beginning (far-left) and perform the same logic. Now, we will look at $n-1$ element, as the last element is in the correct place and we don't need to check it.
- So, the first iteration, took $\mathcal{O}(n)$, the second took $\mathcal{O}(n-1)$, the third took $\mathcal{O}(n-3)$, etc. until getting $\mathcal{O}(3)$, $\mathcal{O}(2)$ and $\mathcal{O}(1)$.
- How many iterations will we have? For each value, there is an iteration associated with it (therefore, n iterations) and the worst iteration took $\mathcal{O}(n)$, hence, the total number of steps is $n \times n = n^2$, or $\mathcal{O}\left(n^2\right)$.

Bubble/Exchange Sort II

- The code is the following:

```java
//compare two adjacent cells together, swap if needed.
//Note: Far-left cell not in outer loop, but visited in inner loop.
for (int i = data.length - 1; i >= 1; i--) {
  for (int j = 0; j < i; j++) {
    if (data[j + 1] < data[j]) {//change < to > to reverse order
      //perform swap
      int temp = data[j];
      data[j] = data[j + 1];
      data[j + 1] = temp;
    }
  }
}
```

- The outer loop is the limit of going from 1 to n, then 1 to $n-1$, then 1 to $n-2$, etc.
- The inner loop will go up to that specified limit and perform the swap if needed.
- Always use a temporary variable when swapping!
- In case the order needs to be reversed, change the `if` statement to `if (data[j + 1] > data[j]) { ... } //change < to >`

Bubble/Exchange Sort III

- Here is an example on how to apply the algorithm on an unsorted list:

2	7	6	5	1	4	9	0	3	8	Raw Data
2	7	6	5	1	4	9	0	3	8	No swap
2	6	7	5	1	4	9	0	3	8	Swap
2	6	5	7	1	4	9	0	3	8	Swap
2	6	5	1	7	4	9	0	3	8	Swap
2	6	5	1	4	7	9	0	3	8	Swap
2	6	5	1	4	7	9	0	3	8	Swap
2	6	5	1	4	7	0	9	3	8	Swap
2	6	5	1	4	7	0	3	9	8	Swap
2	6	5	1	4	7	0	3	8	9	Swap
2	6	5	1	4	7	0	3	8	9	Last value is sorted

- The above was each step of the first iteration. The next slide will have all of the iterations, but without showing the steps of each iteration.

Bubble/Exchange Sort III

- The iterations on how to apply the algorithm on an unsorted list:

2	7	6	5	1	4	9	0	3	8	Raw Data
2	6	5	1	4	7	0	3	8	9	
2	5	1	4	6	0	3	7	8	9	
2	1	4	5	0	3	6	7	8	9	
1	2	4	0	3	5	6	7	8	9	
1	2	0	3	4	5	6	7	8	9	
1	0	2	3	4	5	6	7	8	9	
0	1	2	3	4	5	6	7	8	9	
0	1	2	3	4	5	6	7	8	9	
0	1	2	3	4	5	6	7	8	9	

- Note that the far-left value by itself will always be considered as sorted.

Selection Sort I

- Selection sort will start at the far-left cell and try to swap the value in that cell with the smallest value found. Then, focuses on the next cell, looks for the second smallest value and swapped them, and so on...
- It will remember the cell indices, then swap the values found in those cells.
- For example, suppose we have the following data to start with:

`data`	20	70	60	50	10	40	90	0	30	80
Index	0	1	2	3	4	5	6	7	8	9

- We will start at index 0 and find the smallest value in the array. It happens to be the value 0 at index 7.
- We will remember the two indices 0 and 7 to swap their values:

`data`	0	70	60	50	10	40	90	20	30	80
Index	0	1	2	3	4	5	6	7	8	9

- We now move to index 1 and swap the value with the second smallest in the array, which happens to be 10:

`data`	0	10	60	50	70	40	90	20	30	80
Index	0	1	2	3	4	5	6	7	8	9

- We continue until the end...

Selection Sort II

- The code is the following:

```
int maxValueIndex; // index of the largest item
//far-right cell not in loop, but visited in code
for (int i = 0; i < data.length - 1; i++) {
  maxValueIndex = i;
  for (int j = i + 1; j < data.length; j++) {
    //found a new larger value, store the new larger value index
    if (data[j] < data[maxValueIndex]) {//< or > for reversing order
        maxValueIndex = j;
    }
  }
  //perform the swapping
  int temp = data[i];
  data[i] = data[maxValueIndex];
  data[maxValueIndex] = temp;
}
```

- In case the order needs to be reversed, change the `if` statement to `if (data[j] > data[maxValueIndex]) { ... } //change < to >`

Selection Sort III

- Here is an example on how to apply the algorithm on an unsorted list:

20	70	60	50	10	40	90	0	30	80	Raw Data
0	20	60	50	10	40	90	70	30	80	Index 0 is sorted
0	10	60	50	20	40	90	70	30	80	Indices 0 to 1 are sorted
0	10	20	50	60	40	90	70	30	80	Indices 0 to 2 are sorted
0	10	20	30	60	40	90	70	50	80	Indices 0 to 3 are sorted
0	10	20	30	40	60	90	70	50	80	Indices 0 to 4 are sorted
0	10	20	30	40	50	90	70	60	80	Indices 0 to 5 are sorted
0	10	20	30	40	50	60	70	90	80	Indices 0 to 6 are sorted
0	10	20	30	40	50	60	70	90	80	Indices 0 to 7 are sorted
0	10	20	30	40	50	60	70	80	90	Indices 0 to 8 are sorted

- Again, the last element is sorted by default.

Insertion Sort I

- Insertion sort will start at the second most far-right (*i.e.*, not n^{th} element, but $n-1^{th}$) and then eventually reach the far-left.
- Each iteration, we will try to shift the current value to the right as far as possible. It will shift the elements to make room for the current value to be placed at the appropriate cell.
- The current value will be overridden in the array, which is why we need to temporarily store it inside of a variable. For example, suppose we have the following data to start with:

`data`	20	70	60	50	10	40	90	0	30	80
Index	0	1	2	3	4	5	6	7	8	9

- We start at 30 and realize that it is in order because we cannot move it to the right. We go one cell to the left and try to move 0. We realize we cannot move it as it is in order. We now go to 90. We keep on shifting to the right until it is in the correct place.

Insertion Sort II

- Coding-wise, we store the value 90 in a temporary variable and then shift all of the elements 0, 30 and 80 one cell to the left:

data	20	70	60	50	10	40	0	30	80	
Index	0	1	2	3	4	5	6	7	8	9

- Now, we have free space to place the 90 again to get

data	20	70	60	50	10	40	0	30	80	90
Index	0	1	2	3	4	5	6	7	8	9

- We continue the process until we reach the far-left element and perform shifting it to the right. The code is:

```
//far-right cell not in loop, but visited in code
for (int i = data.length - 2; i >= 0; i--) {
  int current = data[i];//store current cell temporarily
  int j = i + 1;//index of cell to right of current
  //Shift the elements to the left one cell
  while (j < data.length && data[j] < current) {//change < to >
    data[j - 1] = data[j];
    j = j + 1;
  }
  data[j - 1] = current;//place overridden value in j-1 cell
}
```

Insertion Sort III

- Here is an example on how to apply the algorithm on an unsorted list:

70	20	60	50	10	40	90	0	30	80	Raw Data
70	20	60	50	10	40	90	0	30	80	
70	20	60	50	10	40	90	0	30	80	
70	20	60	50	10	40	0	30	80	90	
70	20	60	50	10	0	30	40	80	90	
70	20	60	50	0	10	30	40	80	90	
70	20	60	0	10	30	40	50	80	90	
70	20	0	10	30	40	50	60	80	90	
70	0	10	20	30	40	50	60	80	90	
0	10	20	30	40	50	60	70	80	90	

Merge Sort I

- Merge sort is a recursive algorithm that is in-situ (*i.e.*, requires $\mathcal{O}(n)$, not $\mathcal{O}(C)$ of extra space).
- It uses the idea of *divide-and-conquer* where we partition the size of the main problem into sub-problems, and keep doing the partitioning until we have a very small sub-problems to solve. We have to solve several of these sub-problems to solve the main problem.
- As mentioned in the previous slides, merge sort has $\mathcal{O}(n \log n)$, which is *significantly* faster than $\mathcal{O}\left(n^2\right)$.
- There will be two passes when using merge sort: *partitioning* the elements and *sorting* the partitioned elements.
- The partitioning phase ensures that our data is divided into single elements. We start with n elements, we keep on partitioning until we have individuals elements. Then, we will sort these individual elements and combine them with more elements. We continue on combining the sorted elements with other elements and resort them until we combine all of the n and sort all of the n elements.
- It will have the logic of sorting, which is called at the *very* end of the method.

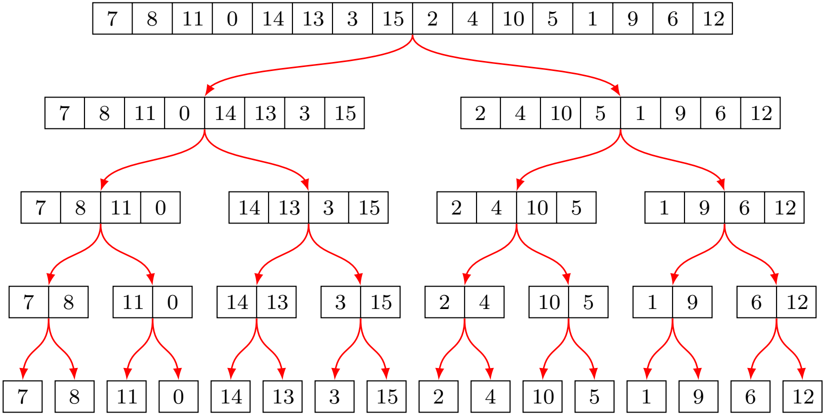
Merge Sort II

- The recursive logic is stopped when we cannot break the elements any more (*i.e.*, when we reach one element).
- A temporary array is used to place the elements once the sorting process begins. It will read the values from the original data and then places values in the temporary array. Lastly, copy the values from the temporary array into the original array to ensure the original array has the sorted value.
- We will partition the array by focusing on specific segments of the array
- The next slide show which elements we will be focusing on once we start sorting. First, we need to make our focus on single elements. Then, we start comparing and sorting.
- We sort the smaller parts first, combine them, sort the large parts, combine them, then sort the largest part at the very end.
- Note that the explanation on the next slide is *different* than how the recursive logic operates which it comes to the order (*i.e.*, which is done first, second, etc.).

Merge Sort III

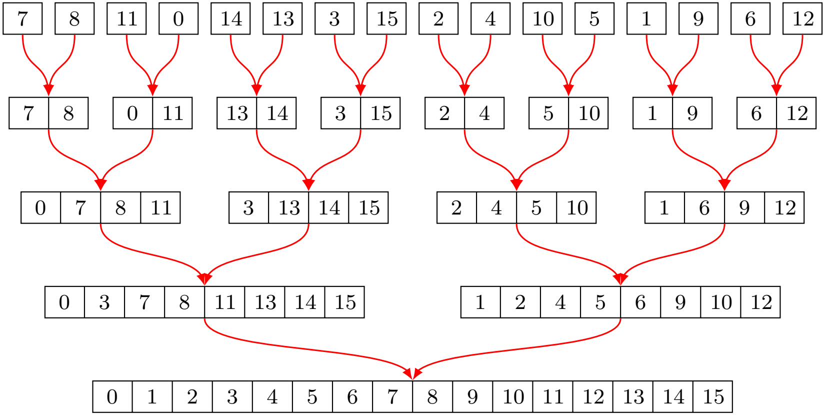
- It is easiest to visualize the algorithm when we have $n = 2^k$ (*e.g.*, $n = 1, 2, 4, 8, 16, 32, 64$, etc.)
- Suppose we have the following $n = 16$ elements to sort:

$$7 \quad 8 \quad 11 \quad 0 \quad 14 \quad 13 \quad 3 \quad 15 \quad 2 \quad 4 \quad 10 \quad 5 \quad 1 \quad 9 \quad 6 \quad 12$$



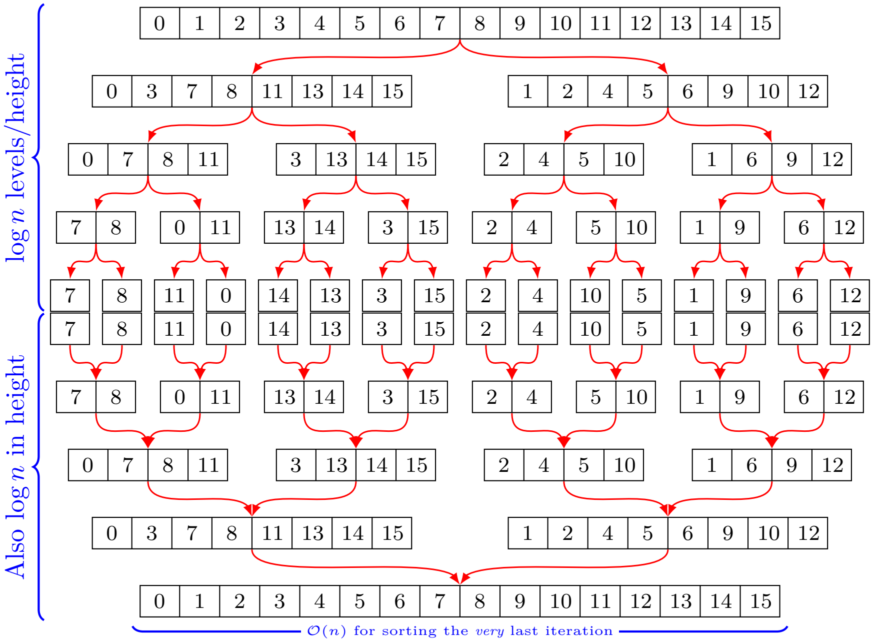
Merge Sort IV

- We have divide all the elements until each partition has one element. We will start sorting! The first is combine the two element and sort them, then four elements and sort them, etc., until we get to complete the last iteration, which has all of the n elements.



- Let use see both the partitioning and sorting on the next slide.

Merge Sort V

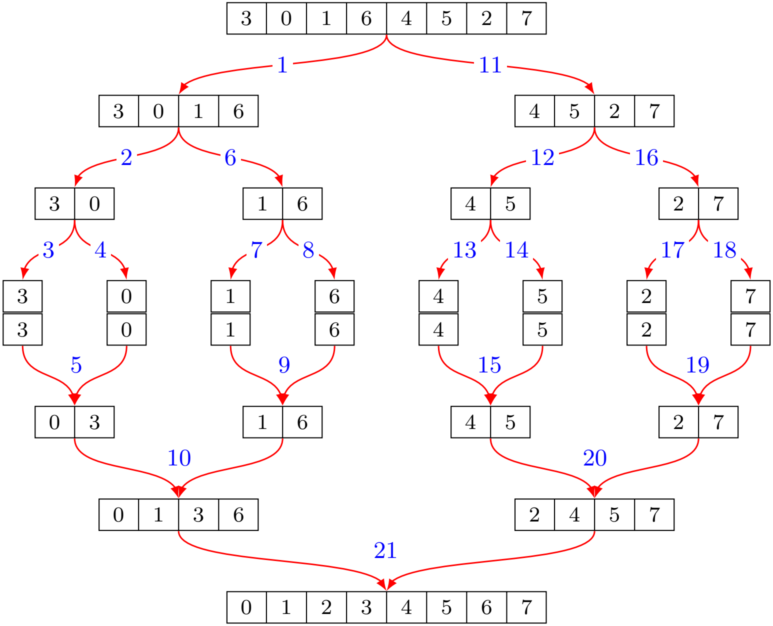


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 3 7 8 11 13 14 15
1 2 4 5 6 9 10 12
0 7 8 11
3 13 14 15
2 4 5 10
1 6 9 12
7 8
0 11
13 14
3 15
2 4
5 10
1 9
6 12
7 8 11 0 14 13 3 15 2 4 10 5 1 9 6 12
7 8 11 0 14 13 3 15 2 4 10 5 1 9 6 12
7 8
0 11
13 14
3 15
2 4
5 10
1 9
6 12
0 7 8 11
3 13 14 15
2 4 5 10
1 6 9 12
0 3 7 8 11 13 14 15
1 2 4 5 6 9 10 12
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
log n levels/height
Also log n in height
$\mathcal{O}(n)$ for sorting the *very* last iteration

Merge Sort VI

- The height, or number of levels required to ensure all of the elements are divided into individual cells is $\log n$.
- The number of levels to sort will also be $\log n$.
- Out of all of the levels, the *very* last level will take $\mathcal{O}(n)$ to be sorted. This is because we will loop through the left sub-array (which has $\frac{n}{2}$ elements) and the right sub-array (which also has $\frac{n}{2}$) and will place the sorted elements in order. In total, we will need to loop through n elements.
- What is the $\mathcal{O}$ notation? It is simple, the number of levels times the level that took the most steps to sort. We know that there are $\log n$ levels, and the level that took the most steps is the last level, which took n steps to sort. Hence, the $\mathcal{O}$ notation of merge sort is $\mathcal{O}(n \log n)$.
- The order is defined as to keep on partitioning and take the left route. Keeping doing that until we can go left any more, then we start going right. This means that we need to backtrack (*i.e.*, go up) and then take the right route. Once we cannot go right any more, we perform the sort.
- The order of the operations is found on the next slide.

Merge Sort VII



Merge Sort VIII

- The recursive method `mergeSort` is simple, and also calls the non-recursive method `mergeLogic`. On the other hand, `mergeLogic` is quite involved.

```java
private void mergeSort(int data[], int temp[], int left, int right) {
  if (left >= right) {// Base case
    //left is the smaller index and right is the larger index,
    //either left and right swapped or current they are equal,
    //which means the partition is one element, stop recursion.
    return;
  }
  int mid = (right + left) / 2;//middle index of the array
  //recursive calls:
  mergeSort(data, temp, left, mid);//recursive left partition
  mergeSort(data, temp, mid + 1, right);//recursive right partition
  //The merge logic method is non-recursive
  mergeLogic(data, temp, left, mid + 1, right);//sort partition
}
```

- The `mergeLogic` method is found on the next slides.

Merge Sort IX

```
private void mergeLogic(
  int data[], // First parameter : original data
  int temp[], // Second parameter: temporary data to use
  int left,   // Third parameter : starting index of the partition
  int mid,    // Fourth parameter: midpoint index of the partition
  int right   // Fifth parameter : ending index of the partition
) {
  int n = right - left + 1;
  int end = mid - 1;
  int i = left;
  while ((left <= end) && (mid <= right)) {//sort both partitions
    if (data[left] <= data[mid]) {
      temp[i] = data[left];
      left = left + 1;
    } else {
      temp[i] = data[mid];
      mid = mid + 1;
    }
    i = i + 1;
  }
  //continue on the next slide ...
```

Merge Sort X

```
    //continuing from the previous slide:
    //place remaining elements from data into LEFT partition of temp.
    while (left <= end) {
      temp[i] = data[left];
      left = left + 1;
      i = i + 1;
    }

    //place remaining elements from data into RIGHT partition of temp.
    while (mid <= right) {
      temp[i] = data[mid];
      mid = mid + 1;
      i = i + 1;
    }

    //Place entire data from the temp array into the original array.
    for (int z = 0; z < n; z++) {
      data[right] = temp[right];
      right = right - 1;
    }
  } // The ending brace of the mergeLogic method from previous slide.
```