
Recursion

Reading:
Savitch ch. 11

Introduction

- ♦ Sometimes it is useful to define a method in terms of itself.
- ♦ A method definition is *recursive* if it contains a call to itself.
- ♦ The method continues to call itself with ever simpler cases, until a *base case* is reached which can be resolved without any recursive calls.

Example – Searching a phone book

- ♦ Open the book to the middle.
- ♦ If the name is on that page, you're done.
- ♦ If the name comes before the names on the page, use the same approach to search for the name in the first half of the phone book.
- ♦ Otherwise use the same approach to search for the name in the second half of the phone book.

Case Study – Digits to Words

Savitch p 783

- ♦ Write a method definition **displayAsWords** that accepts a single **int** and produces words representing its digits.
- ♦ Example:
Input: **223**
Output: **two two three**
- ♦ Recursive algorithm
 - output all but the last digit as words
 - output the word for the last digit

Case Study – Digits to Words

- ♦ Use a helper method **getWordFromDigit** to get the word for a number < 10 (“one”, “two”, ...).
- ♦ To “chop off” the last digit, use integer division by 10
 $(327 / 10) // 32$
- ♦ To get just the last digit, use $\% 10$
 $(327 \% 10) // 7$
- ♦ See **RecursionDemo.java** under SavitchSrc on the course website

Case Study – Digits to Words

- ♦ The **base case** is when number is a single digit (**number < 10**).
- ♦ In this case, we just print the String returned by **getWordFromDigit**

```
System.out.print(getWordFromDigit(number) + " ");
```

Case Study – Digits to Words

- ◆ If (**number** **>= 10**) we call **displayAsWords** with all except the last digit, then print the last digit.

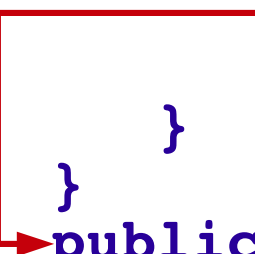
```
displayAsWords(number/10); // last digit removed  
System.out.print(getWordFromDigit(number%10) + " ");
```

How Recursion Works

- ◆ Nothing special is required to handle a call to a recursive method.
- ◆ At each call, the needed arguments are provided and the code is executed.
- ◆ When the method completes, control returns to the statement following the call to the method.
- ◆ See example (987) on the following slides


```
public void displayAsWords(int 987) {  
    if (987 < 10) {  
        System.out.print(getWordFromDigit(987) + " ");  
    } else {  
        98  
        displayAsWords(987/10);  
        System.out.print(getWordFromDigit(987%10) + " ");  
    }  
}
```

```
public void displayAsWords(int 987) {  
    if (987 < 10) {  
        System.out.print(getWordFromDigit(987) + " ");  
    } else {  
        98  
        displayAsWords(987/10);  
        System.out.print(getWordFromDigit(987%10) + " ");  
    }  
}  
→ public void displayAsWords(int 98) {  
    if (98 < 10) {  
        System.out.print(getWordFromDigit(98) + " ");  
    } else {  
        9  
        displayAsWords(98/10);  
        System.out.print(getWordFromDigit(98%10) + " ");  
    }  
}
```



```
public void displayAsWords(int 987) {
    if (987 < 10) {
        System.out.print(getWordFromDigit(987) + " ");
    } else {
        98
        displayAsWords(987/10);
        System.out.print(getWordFromDigit(987%10) + " ");
    }
}
→ public void displayAsWords(int 98) {
    if (98 < 10) {
        System.out.print(getWordFromDigit(98) + " ");
    } else {
        9
        displayAsWords(98/10);
        System.out.print(getWordFromDigit(98%10) + " ");
    }
}
→ public void displayAsWords(int 9) {
    if (9 < 10) {
        System.out.print(getWordFromDigit(9) + " ");
    } else {
        displayAsWords(9/10);    nine
        System.out.print(getWordFromDigit(9%10) + " ");
    }
}
```

```
public void displayAsWords(int 987) {
    if (987 < 10) {
        System.out.print(getWordFromDigit(987) + " ");
    } else {
        98
        displayAsWords(987/10);
        System.out.print(getWordFromDigit(987%10) + " ");
    }
}
→ public void displayAsWords(int 98) {
    if (98 < 10) {
        System.out.print(getWordFromDigit(98) + " ");
    } else {
        9
        displayAsWords(98/10);
        System.out.print(getWordFromDigit(98%10) + " ");
    }
}
→ public void displayAsWords(int 9) {
    if (9 < 10) {
        System.out.print(getWordFromDigit(9) + " ");
    } else {
        displayAsWords(9/10);
        System.out.print(getWordFromDigit(9%10) + " ");
    }
}
```

The diagram illustrates the recursive process of converting the number 987 into words. Red arrows show the sequence of recursive calls: `displayAsWords(987)` calls `displayAsWords(98)`, which in turn calls `displayAsWords(9)`. Red brackets and labels indicate the return values from the recursive calls: the call `displayAsWords(98)` returns the string `"eight"` (from `98%10`), and the call `displayAsWords(9)` returns the string `"nine"` (from `9%10`).

```
public void displayAsWords(int 987) {  
    if (987 < 10) {  
        System.out.print(getWordFromDigit(987) + " ");  
    } else {  
        98  
        displayAsWords(987/10);  
        7  
        System.out.print(getWordFromDigit(987%10) + " ");  
    }  
}  
→ public void displayAsWords(int 98) {  
    if (98 < 10) {  
        System.out.print(getWordFromDigit(98) + " ");  
    } else {  
        9  
        displayAsWords(98/10);  
        8  
        System.out.print(getWordFromDigit(98%10) + " ");  
    }  
}  
→ public void displayAsWords(int 9) {  
    if (9 < 10) {  
        System.out.print(getWordFromDigit(9) + " ");  
    } else {  
        displayAsWords(9/10);  
        nine  
        System.out.print(getWordFromDigit(9%10) + " ");  
    }  
}
```

The diagram illustrates the recursive process of converting the number 987 to words. Red arrows trace the sequence of recursive calls from the initial call `displayAsWords(987)` down to `displayAsWords(9)` and back up. Brackets and labels indicate the return values of the recursive calls: `displayAsWords(987)` returns "seven", `displayAsWords(98)` returns "eight", and `displayAsWords(9)` returns "nine".

Recursion Guidelines

- ♦ The definition of a recursive method typically includes an **if-else** statement.
 - ♦ One branch represents a **base case** which can be solved directly – without recursion.
 - ♦ Another branch includes a **recursive call** to the method, but with simpler or smaller arguments.
- ♦ The base case must be reached eventually.

Recursion Guidelines

- ♦ If the recursive call inside the method does not use a smaller or simpler argument, the base case may never be reached.
- ♦ The method then calls itself forever (or until resources run out).
- ♦ This is called *infinite recursion*.

Recursion vs. Iteration

- ♦ Any recursive method can be written without using recursion.
- ♦ Typically this is done with a loop.
- ♦ The resulting method is called the *iterative version*.
- ♦ See **RecursionDemo.java** and **IterativeDemo.java** under SavitchSrc on the course website

Recursion vs. Iteration

- ♦ A recursive version of a method typically executes less efficiently than the iterative version.
- ♦ This is because the computer must keep track of the recursive calls and the suspended computations.
- ♦ However, sometimes it is much easier to write a method recursively than iteratively.

Exercise 1

Self-Test Question 1, page 792. What is the output?

```
public class RecursionExercise {  
    public static void main(String[] args) {  
        methodA(3);  
    }  
  
    public static void methodA(int n) {  
        if (n < 1)  
            System.out.println("B");  
        else {  
            methodA(n - 1);  
            System.out.println("R");  
        }  
    }  
}
```

```
public static void methodA(int 3) {
    if (3 < 1) System.out.println("B");
    else {
        methodA(3 - 1);
        System.out.println("R");
    }
}
→ public static void methodA(int 2) {
    if (2 < 1) System.out.println("B");
    else {
        methodA(2 - 1);
        System.out.println("R");
    }
}
→ public static void methodA(int 1) {
    if (1 < 1) System.out.println("B");
    else {
        methodA(1 - 1);
        System.out.println("R");
    }
}
→ public static void methodA(int 0) {
    if (0 < 1) System.out.println("B"); → B
    else {
        methodA(0 - 1);
        System.out.println("R");
    }
}
```

Output:
B

recursion

```

public static void methodA(int 3) {
    if (3 < 1) System.out.println("B");
    else {
        methodA(3 - 1);
        System.out.println("R");
    }
}
→ public static void methodA(int 2) {
    if (2 < 1) System.out.println("B");
    else {
        methodA(2 - 1);
        System.out.println("R");
    }
}
→ public static void methodA(int 1) {
    if (1 < 1) System.out.println("B");
    else {
        methodA(1 - 1);
        System.out.println("R"); → R
    }
}
→ public static void methodA(int 0) {
    if (0 < 1) System.out.println("B"); → B
    else {
        methodA(0 - 1);
        System.out.println("R");
    }
}

```

Output:
B
R

unwing

```
public static void methodA(int 3) {  
    if (3 < 1) System.out.println("B");  
    else {  
        methodA(3 - 1);  
        System.out.println("R");  
    }  
}  
→ public static void methodA(int 2) {  
    if (2 < 1) System.out.println("B");  
    else {  
        methodA(2 - 1);  
        System.out.println("R"); → R  
    }  
}  
→ public static void methodA(int 1) {  
    if (1 < 1) System.out.println("B");  
    else {  
        methodA(1 - 1);  
        System.out.println("R"); → R  
    }  
}  
→ public static void methodA(int 0) {  
    if (0 < 1) System.out.println("B"); → B  
    else {  
        methodA(0 - 1);  
        System.out.println("R");  
    }  
}
```

Output:
B
R
R

unwing

```
public static void methodA(int 3) {  
    if (3 < 1) System.out.println("B");  
    else {  
        methodA(3 - 1);  
        System.out.println("R");  
    }  
}  
public static void methodA(int 2) {  
    if (2 < 1) System.out.println("B");  
    else {  
        methodA(2 - 1);  
        System.out.println("R");  
    }  
}  
public static void methodA(int 1) {  
    if (1 < 1) System.out.println("B");  
    else {  
        methodA(1 - 1);  
        System.out.println("R");  
    }  
}  
public static void methodA(int 0) {  
    if (0 < 1) System.out.println("B");  
    else {  
        methodA(0 - 1);  
        System.out.println("R");  
    }  
}
```

Output:


**B
R
R
R**

unwing

Exercise 2

What is the output?

```
public class RecursionExercise2 {  
    public static void main(String[] args) {  
        methodB(3);  
    }  
  
    public static void methodB(int n) {  
        if (n < 1)  
            System.out.println("done");  
        else {  
            System.out.println(n);  
            methodB(n - 1);  
        }  
    }  
}
```

A diagram consisting of two red curved arrows. The first arrow starts at the 'methodB(n - 1);' line and points to the 'System.out.println(n);' line above it. The second arrow starts at the 'System.out.println(n);' line and points to the 'methodB(n - 1);' line below it, illustrating the recursive call and return sequence.

```

public static void methodB(int 3) {
    if (3 < 1) System.out.println("done");
    else {
        System.out.println(3); → 3
        methodB(3 - 1);
    }
}
→ public static void methodB(int 2) {
    if (2 < 1) System.out.println("done");
    else {
        System.out.println(2); → 2
        methodB(2 - 1);
    }
}
→ public static void methodB(int 1) {
    if (1 < 1) System.out.println("done");
    else {
        System.out.println(1); → 1
        methodB(1 - 1);
    }
}
→ public static void methodB(int 0) {
    if (0 < 1) System.out.println("done"); → done
    else {
        System.out.println(0);
        methodB(0 - 1);
    }
}

```

在这个递归过程中，没有需要在递归返回后执行的代码，因此每次递归调用完成后就直接返回到上一层调用，而不会执行额外的步骤。

Output:
3
2
1
done

only recursion.
no unwinding.

Returning a Value

- ◆ So far, our recursive methods have been void methods.
- ◆ Recursive methods can also return a value.
- ◆ The value returned by the recursive method call is typically used to compute the return value of the method.
- ◆ Example: Consider a method that takes an **int** argument and returns the number of zeros in the argument. 获取输入的int有多少个0

Returning a Value

- ♦ Do this by getting the number of zeros contained in all but the last digit, then add 1 if the last digit is a zero.
- ♦ For example, the number of zeros in 50020 is the number of zeros in 5002 plus 1 for the last zero.
- ♦ The number of zeros in 50022 is the number of zeros in 5002 without adding 1 because the last digit is not zero.

Returning a Value

- ♦ Algorithm for determining the number of zeros in an int
 - ♦ If **$n < 10$** , return the number of zeros in n (base case)
 - ♦ if **$n == 0$** , return **1**
 - ♦ Otherwise return **0**
 - ♦ Otherwise (recursive invocation)
 - ♦ Get the number of zeros in n with the last digit removed (**$\text{numberOfZeros}(n/10)$**)
 - ♦ Determine if the last digit is zero (**$n\%10 == 0$**)
 - ♦ If yes, return **$\text{numberOfZeros}(n/10) + 1$**
 - ♦ If no, return **$\text{numberOfZeros}(n/10)$**

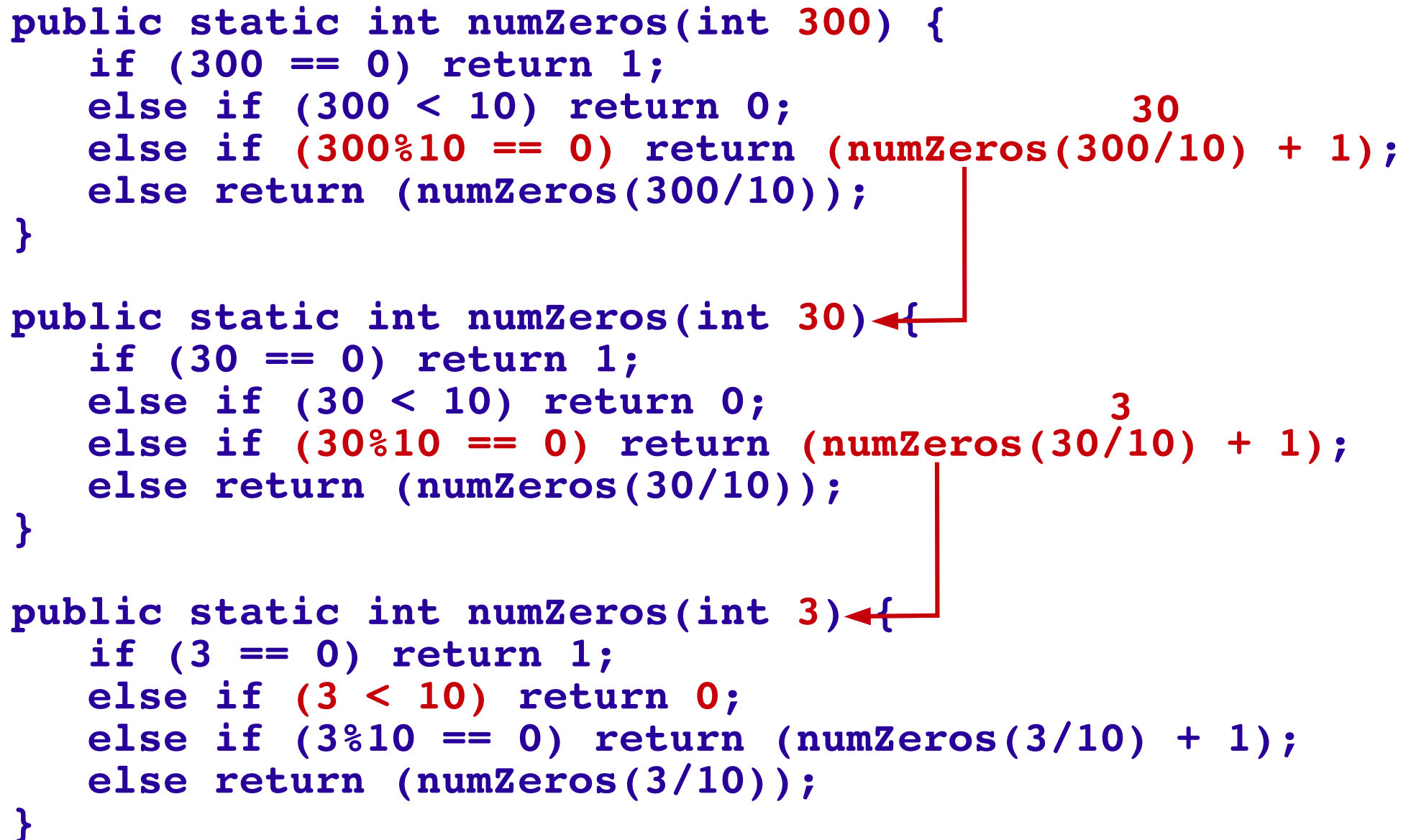
Returning a Value

```
public static int numZeros(int n) {  
    if (n == 0) {  
        //n has one digit that is 0 (base case)  
        return 1;  
    } else if (n < 10) {  
        //n has one digit that is not 0 (base case)  
        return 0;  
    } else if (n%10 == 0) {  
        //last digit is 0 (recursive call)  
        return (numZeros(n/10) + 1);  
    } else {  
        //last digit is not 0 (recursive call)  
        return (numZeros(n/10));  
    }  
}
```

- See example $n = 300$ on the following slides

Returning a Value

```
public static int numZeros(int 300) {  
    if (300 == 0) return 1;  
    else if (300 < 10) return 0;  
    else if (300%10 == 0) return (numZeros(300/10) + 1);  
    else return (numZeros(300/10));  
}  
  
public static int numZeros(int 30) {  
    if (30 == 0) return 1;  
    else if (30 < 10) return 0;  
    else if (30%10 == 0) return (numZeros(30/10) + 1);  
    else return (numZeros(30/10));  
}  
  
public static int numZeros(int 3) {  
    if (3 == 0) return 1;  
    else if (3 < 10) return 0;  
    else if (3%10 == 0) return (numZeros(3/10) + 1);  
    else return (numZeros(3/10));  
}
```



Returning a Value

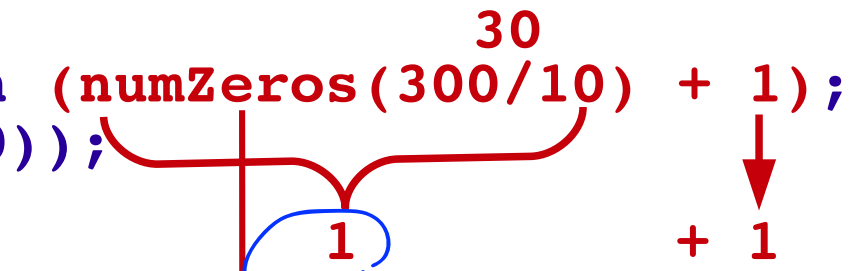
```
public static int numZeros(int 300) {  
    if (300 == 0) return 1;  
    else if (300 < 10) return 0;  
    else if (300%10 == 0) return (numZeros(300/10) + 1);  
    else return (numZeros(300/10));  
}
```

```
public static int numZeros(int 30) {  
    if (30 == 0) return 1;  
    else if (30 < 10) return 0;  
    else if (30%10 == 0) return (numZeros(30/10) + 1);  
    else return (numZeros(30/10));  
}
```

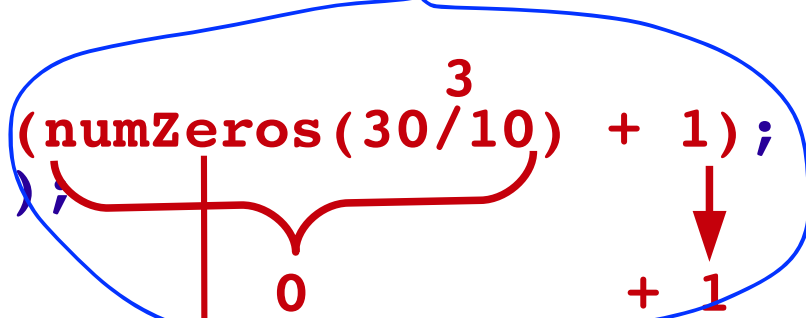
```
public static int numZeros(int 3) {  
    if (3 == 0) return 1;  
    else if (3 < 10) return 0;  
    else if (3%10 == 0) return (numZeros(3/10) + 1);  
    else return (numZeros(3/10));  
}
```

Returning a Value

```
public static int numZeros(int 300) {  
    if (300 == 0) return 1;  
    else if (300 < 10) return 0;  
    else if (300%10 == 0) return (numZeros(300/10) + 1);  
    else return (numZeros(300/10));  
}
```



```
public static int numZeros(int 30) {  
    if (30 == 0) return 1;  
    else if (30 < 10) return 0;  
    else if (30%10 == 0) return (numZeros(30/10) + 1);  
    else return (numZeros(30/10));  
}
```



```
public static int numZeros(int 3) {  
    if (3 == 0) return 1;  
    else if (3 < 10) return 0;  
    else if (3%10 == 0) return (numZeros(3/10) + 1);  
    else return (numZeros(3/10));  
}
```

Exercise 3

Write a recursive method to reverse the input string:

```
public static String reverse(String s) {  
    // base case: string is length 0 or 1  
    if (s.length() <= 1)  
        return s;  
  
    // reverse all but the first character  
    // then add the first character to the end  
    return reverse(s.substring(1))  
        + s.charAt(0);  
}
```

substring(int start)
substring(int start, int end)

- See example s = "step" on the following slides


```
public static String reverse(String "step") {
    if ("step".length() <= 1)
        return "step";
    return reverse("step".substring(1)) + "step".charAt(0);
}
```

"tep"

```
public static String reverse(String "tep") {
    if ("tep".length() <= 1)
        return "tep";
    return reverse("tep".substring(1)) + "tep".charAt(0);
}
```


"ep"

```
public static String reverse(String "ep") {
    if ("ep".length() <= 1)
        return "ep";
    return reverse("ep".substring(1)) + "ep".charAt(0);
}
```


"p"

```
public static String reverse(String "p") {
    if ("p".length() <= 1)
        return "p";
    return reverse("p".substring(1)) + "p".charAt(0);
}
```

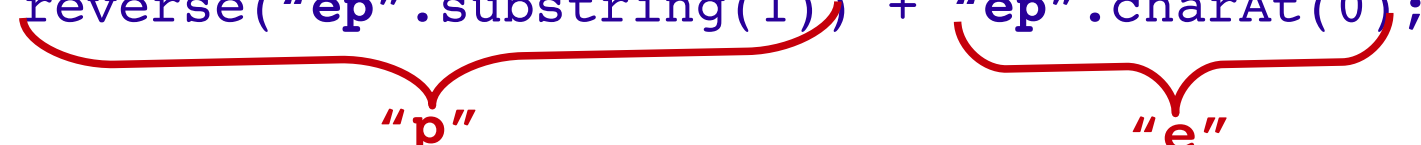
```
public static String reverse(String "step") {  
    if ("step".length() <= 1)  
        return "step";  
    return reverse("step".substring(1)) + "step".charAt(0);  
}
```




```
public static String reverse(String "tep") {  
    if ("tep".length() <= 1)  
        return "tep";  
    return reverse("tep".substring(1)) + "tep".charAt(0);  
}
```



```
public static String reverse(String "ep") {  
    if ("ep".length() <= 1)  
        return "ep";  
    return reverse("ep".substring(1)) + "ep".charAt(0);  
}
```



```
public static String reverse(String "p") {  
    if ("p".length() <= 1)  
        return "p";  
    return reverse("p".substring(1)) + "p".charAt(0);  
}
```


unwinding

Binary Search

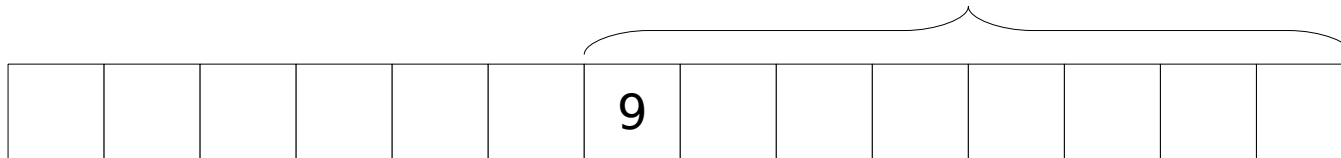
Savitch p 800 (case study)

- ♦ Let's design a recursive method that determines if a given number is in a sorted array of ints.
看int是否在(int)list中,
在, 返回index in the list
不在, 返回-1
- ♦ If the number is in the array, return the index of the number in the array.
- ♦ If the number is not in the array, return -1
- ♦ Instead of searching the array linearly/sequentially (starting at index 0, comparing each element to number), we will search for the number recursively.

Binary Search

- ◆ Since the array is **sorted**, we can rule out entire sections of the array as we search.
- ◆ For example, if we are looking for a 7 and we encounter an element of the array containing a 9, we know that the 7 will not be at any index \geq to the location of the 9.

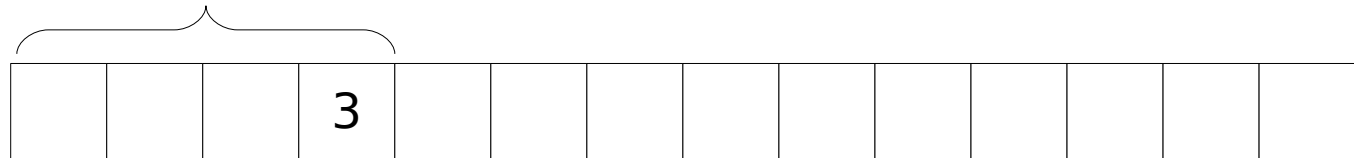
7 can't be here



Binary Search

- Similarly, if we are looking for a 7 and we encounter an element of the array containing a 3, we know that the 7 will not be at any index \leq to the location of the 3.

7 can't be here



- Of course, if we are looking for a 7 and find it, then we stop searching.

Binary Search

- ♦ Binary search is an example of a *divide and conquer* algorithm.
- ♦ We divide the problem into smaller and smaller problems until it can be solved.
- ♦ We use a binary search algorithm when looking up a number in the phone book.
- ♦ Flip open the phone book to a page near the middle.
- ♦ To find the middle of our array, use integer division: $\text{mid} = (\text{first} + \text{last}) / 2$

Binary Search

与在中间index的数字 进行比较

- ♦ **Base case**: If our number is at index **mid** we are done
- ♦ **Recursive case**: If our number is smaller than the number at index **mid**, search the first half (from **first** to **mid-1**).
- ♦ **Recursive case**: If our number is greater than the number at index **mid**, search the second half (from **mid+1** to **last**).
- ♦ See **ArraySearcher.java** under SavitchSrc on the course website

Binary Search – Number Found

- Example: search target 33 in sorted array of length 10 (index 0-9)

search(33, 0, 9) mid=4

33>32, so 5 (mid+1) ~ 9

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9

search(33, 5, 9) mid=7

33<54, so 4 ~ 6 (mid-1)

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9

search(33, 5, 6) mid=5
found at index 5
return 5;

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9
first				mid				last	

$$(0+9)/2=4$$

```

private int search(target 33, first 0, last 9) {
    int mid, result=-1;
    if (first <= last) {
        mid = (first + last)/2;  → mid = (0+9)/2 = 4
        if (target == a[mid]) {
            result = mid;
        } else if (target < a[mid]) {
            result = search(target, first, mid-1);
        } else {
            result = search(target, mid+1, last);
                          33,      5,      9
        }
    }
    return result;
}

```

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9
					first		mid		last
					$(5+9)/2=7$				

0	1	2	3	4	5	6	7	8	9
					first		mid		last
					$(5+9)/2=7$				

```
private int search(target 33, first 5, last 9) {
    int mid, result=-1;
    if (first <= last) {
        mid = (first + last)/2;  → mid = (5+9)/2 = 7
        if (target == a[mid]) {
            result = mid;
        } else if (target < a[mid]) {
            result = search(target, first, mid-1);
        } else {
            result = search(target, mid+1, last);
        }
    }
    return result;
}
```

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9

first last

 mid

 $(5+6)/2=5$

```

private int search(target 33, first 5, last 6) {
    int mid, result=-1;
    if (first <= last) {
        mid = (first + last)/2;  —————> mid = (5+6)/2 = 5
        if (target == a[mid]) { //base case
            result = mid;
        } else if (target < a[mid]) {
            result = search(target, first, mid-1);
        } else {
            result = search(target, mid+1, last);
        }
    }
    return result;
}

```

5

Binary Search – Number Not Found

- ♦ What if the number is not in the array?
- ♦ Eventually, **first** will become greater than **last**.
- ♦ **-1** is returned, indicating that the number was not found.

Binary Search – Number Not Found

- Example: search target 35 in sorted array of length 10 (index 0-9)

search(35, 0, 9) **mid=4**

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9

search(35, 5, 9) **mid=7**

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9

search(35, 5, 6) **mid=5**

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9

search(35, 6, 6) **mid=6**

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9

search(35, 6, 5)
first>last, return -1

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9

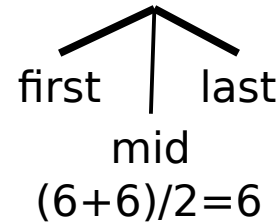
\swarrow \downarrow
 mid first last
 $(5+6)/2=5$

```

private int search(target 35, first 5, last 6) {
    int mid, result=-1;
    if (first <= last) {
        mid = (first + last)/2;       $\longrightarrow$       mid = (5+6)/2 = 5
        if (target == a[mid]) {
            result = mid;
        } else if (target < a[mid]) {
            result = search(target, first, mid-1);
        } else {
            result = search(target, 35, 6, 6, last);
        }
    }
    return result;
}

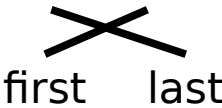
```

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9



```
private int search(target 35, first 6, last 6) {
    int mid, result=-1;
    if (first <= last) {
        mid = (first + last)/2;  → mid = (6+6)/2 = 6
        if (target == a[mid]) {
            result = mid;
        } else if (target < a[mid]) {
            result = search(target, first, mid-1);
                        35,           6,           5
        } else {
            result = search(target, mid+1, last);
        }
    }
    return result;
}
```

5	7	9	13	32	33	42	54	56	88
0	1	2	3	4	5	6	7	8	9


first last

```
private int search(target 35, first 6, last 5) {  
    int mid, result=-1;  
    if (first <= last) {  
        mid = (first + last)/2;  
        if (target == a[mid]) {  
            result = mid;  
        } else if (target < a[mid]) {  
            result = search(target, first, mid-1);  
        } else {  
            result = search(target, mid+1, last);  
        }  
    }  
    return result;  
}
```


Binary Search - Code

- ◆ Notice that the recursive search method is private.
- ◆ A non-recursive, public method **find** is used to get the recursion going.
- ◆ **find** calls search with the first (**0**) and last (**a.length-1**) index of the array.
- ◆ Subsequent recursive calls to search are made with ever smaller portions of the array.

Binary Search - Efficiency

- ♦ With each recursive call, about half of the array is eliminated from consideration.
- ♦ The number of recursive calls required to find an element or determine that it is not in the array is $\log_2 n$ for an array of length n .
- ♦ We say that the binary search algorithm has order $\log_2 n$, written **$O(\log_2 n)$** .
- ♦ This is also known as **big O notation**.

Binary Search - Efficiency

- ♦ For example, an array with 1024 elements will need to do only 10 comparisons.
 - ♦ $\log_2(1024) = 10$
 - ♦ $2^{10} = 1024$

Binary Search - Efficiency

- ♦ An array with 100,000 elements would only need to make about 17 comparisons.
- ♦ This is much better than a linear/sequential search, which would need to make 50,000 comparisons (on average).

Merge Sort

Savitch p 808 (case study)

- ♦ The binary search algorithm works only if the array is sorted.
- ♦ **Merge sort** is a very efficient, recursive algorithm to sort an array.
- ♦ Like binary search, merge sort also takes a “divide and conquer” approach.
 - ♦ The array is divided in halves and the halves are sorted recursively.
 - ♦ Sorted subarrays are merged to form a larger sorted array.

Merge Sort - Pseudocode

Base case: If the array **a** has only 1 element
(**a.length == 1**)

stop, **a** is of length 1, so it is already sorted

Otherwise (recursive case):

Divide the input array **a** into two halves

- copy the first half of the elements into an array called **front**
- copy the second half of the elements into an array called **tail**

Sort array **front** recursively

Sort array **tail** recursively

Merge the arrays **front** and **tail** into **a**

Merging Sorted Arrays - Pseudocode

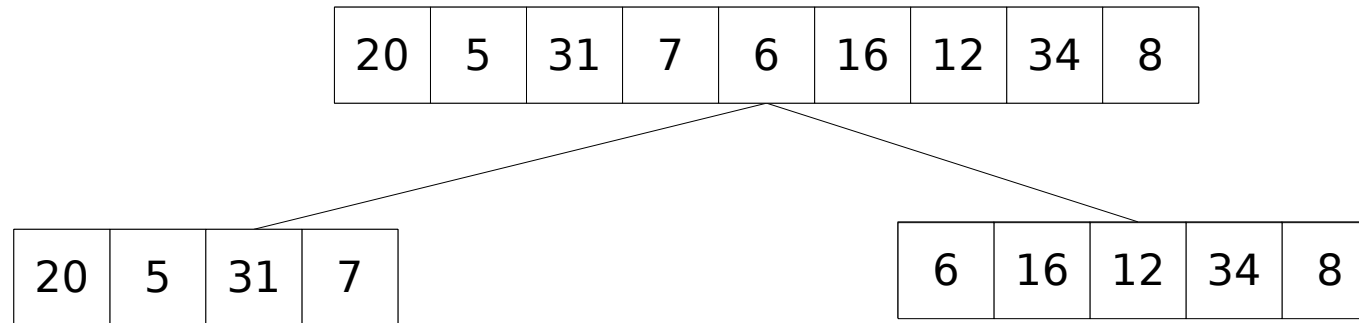
- ♦ **front** and **tail** are both sorted
- ♦ Initialize **frontIndex=0**, **tailIndex=0**, and **aIndex=0**
- ♦ while we haven't reached the end of either list {
 - ♦ copy the smaller of **front** and **tail** to **a**
 - ♦ increment **aIndex**
 - ♦ increment index of array copied from
- ♦ }
- ♦ copy remaining elements from **front**, if any
- ♦ copy remaining elements from **tail**, if any

Merge Sort Visualization

- ♦ Array to sort:

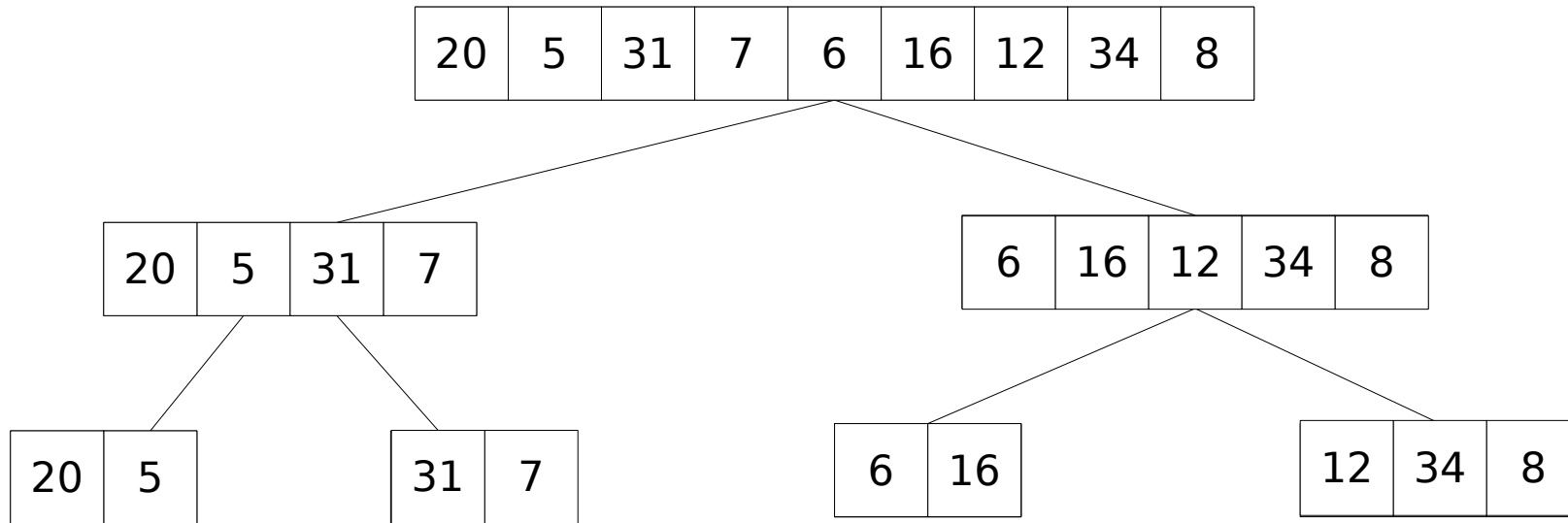
20	5	31	7	6	16	12	34	8
----	---	----	---	---	----	----	----	---

Merge Sort Visualization



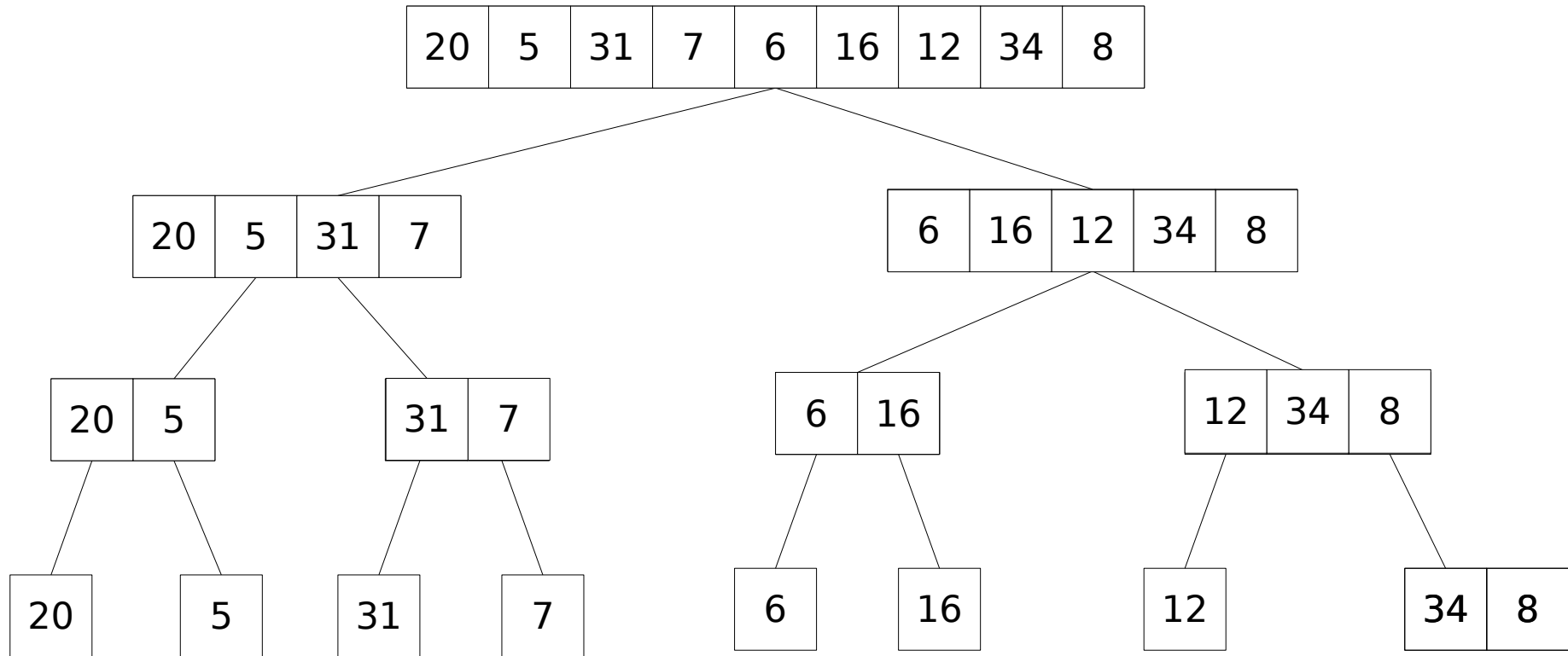
- First, split array(s) recursively into two halves

Merge Sort Visualization



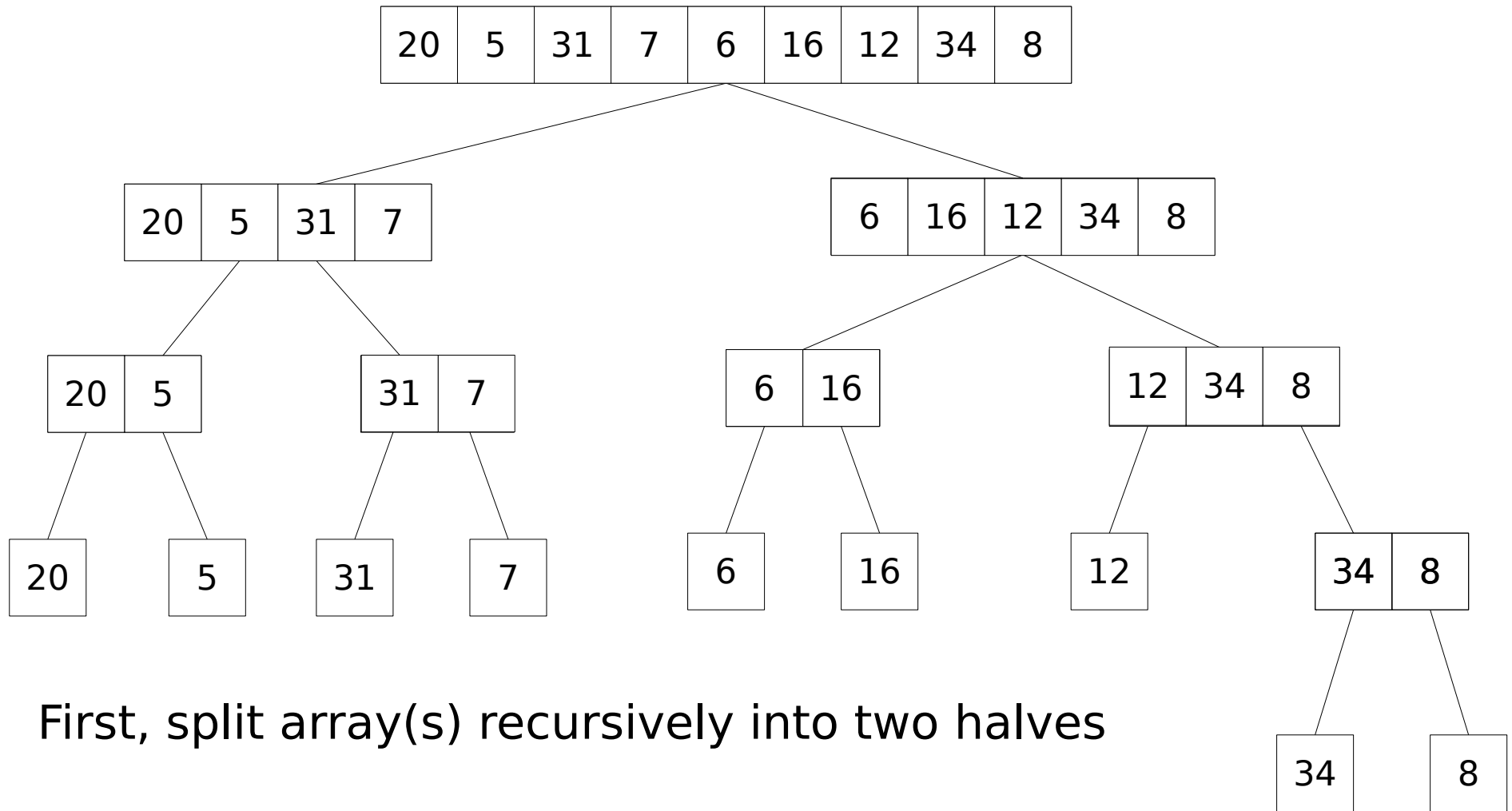
- First, split array(s) recursively into two halves

Merge Sort Visualization

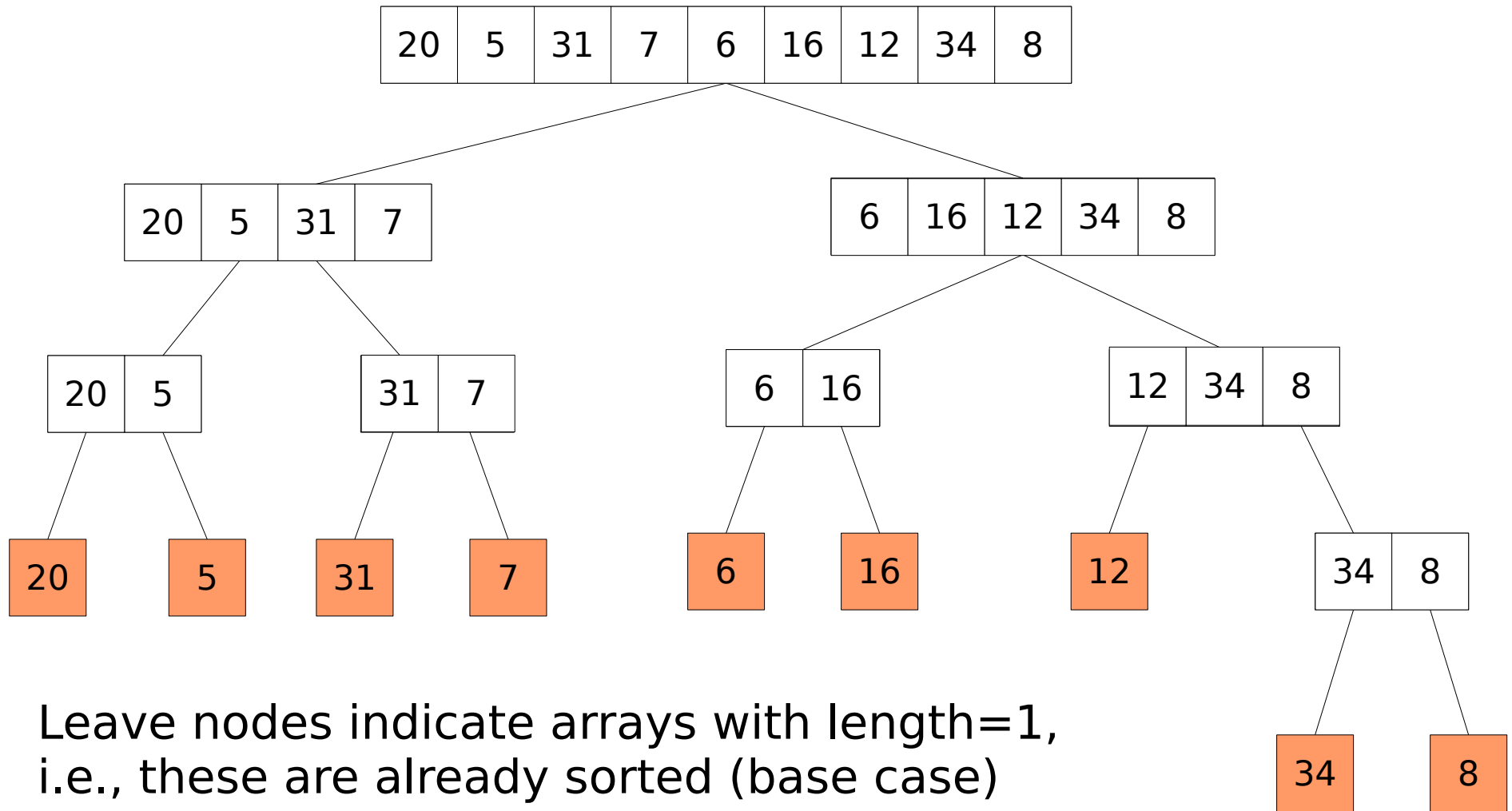


- First, split array(s) recursively into two halves

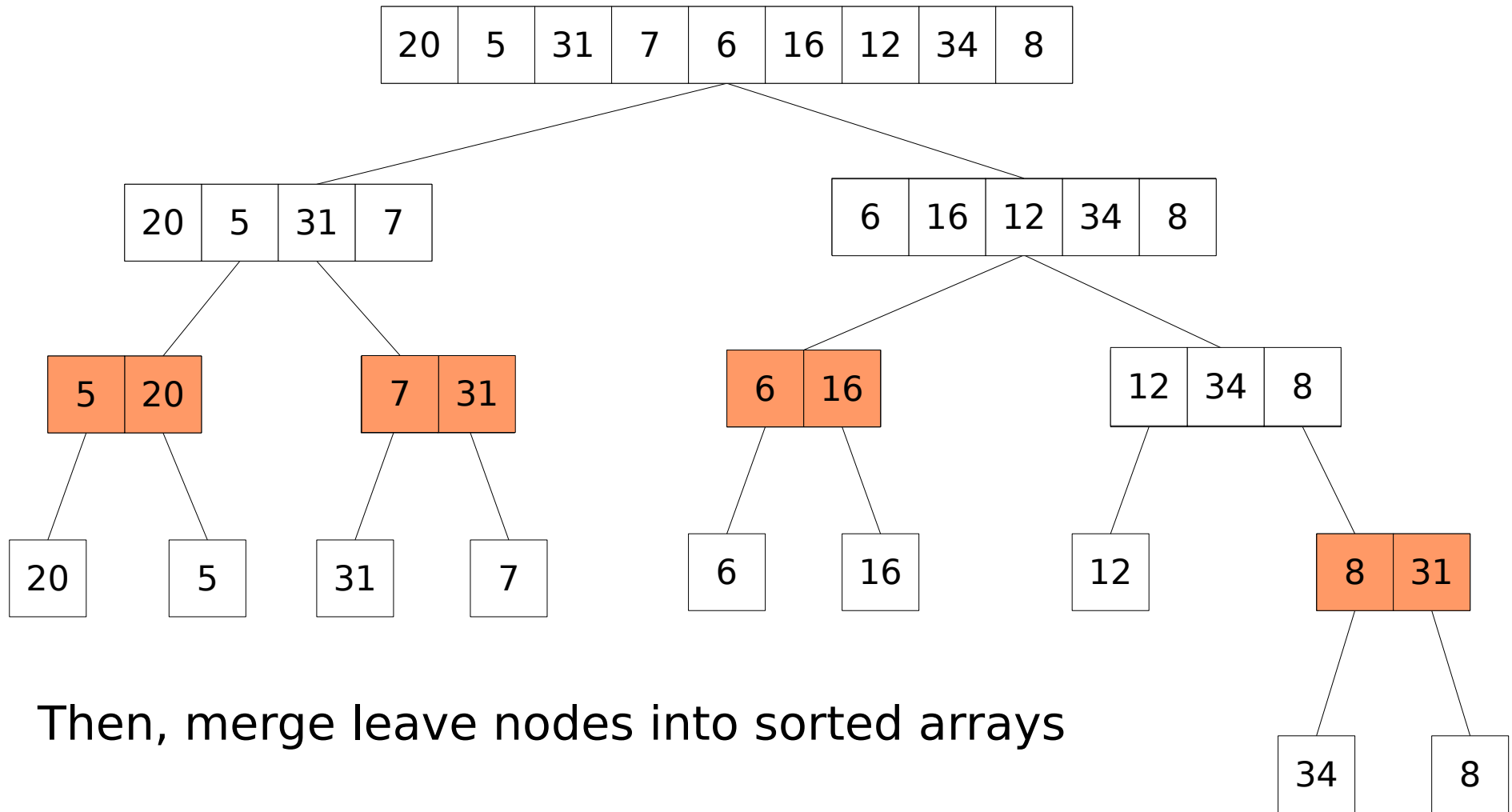
Merge Sort Visualization



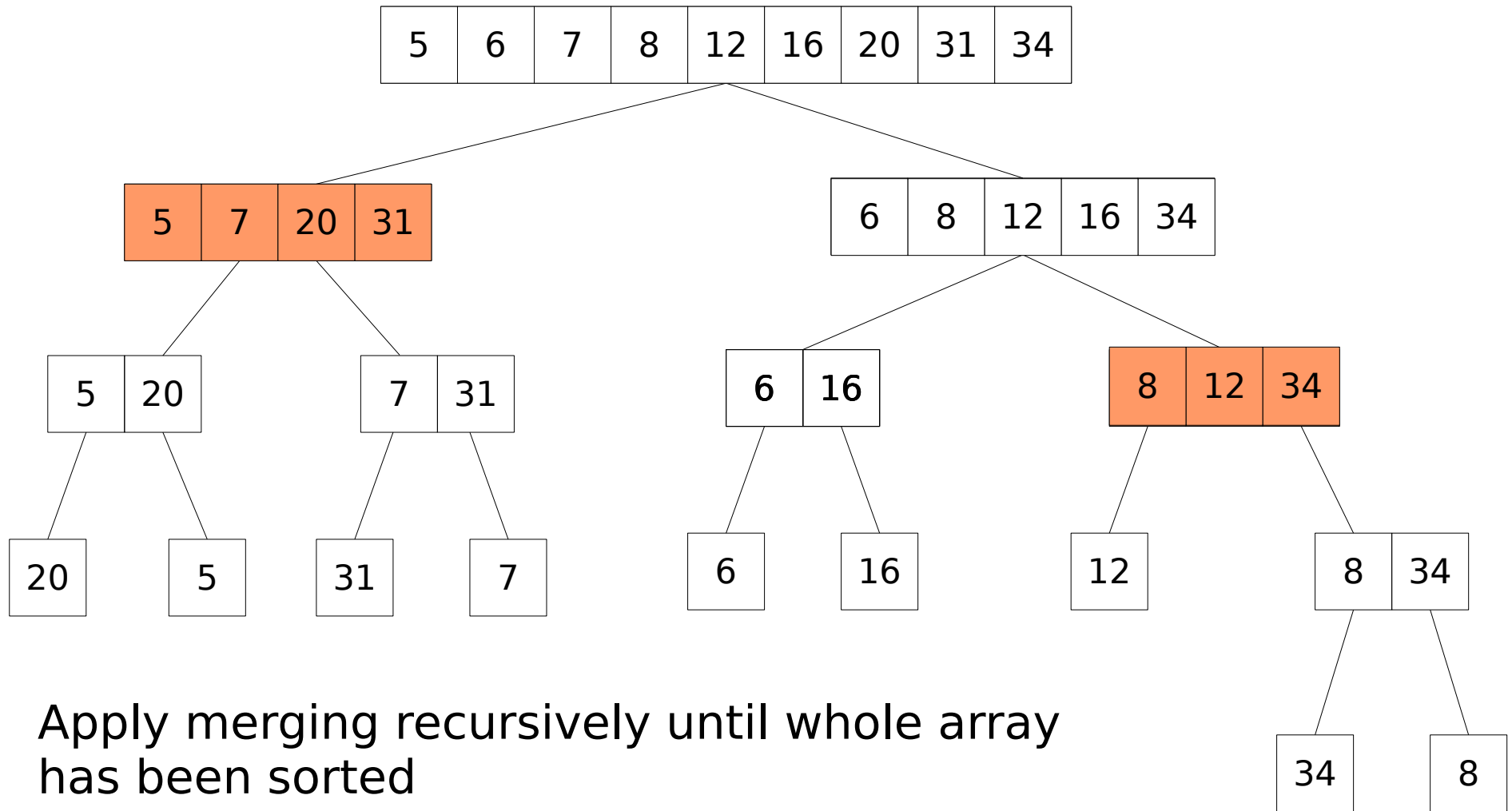
Merge Sort Visualization



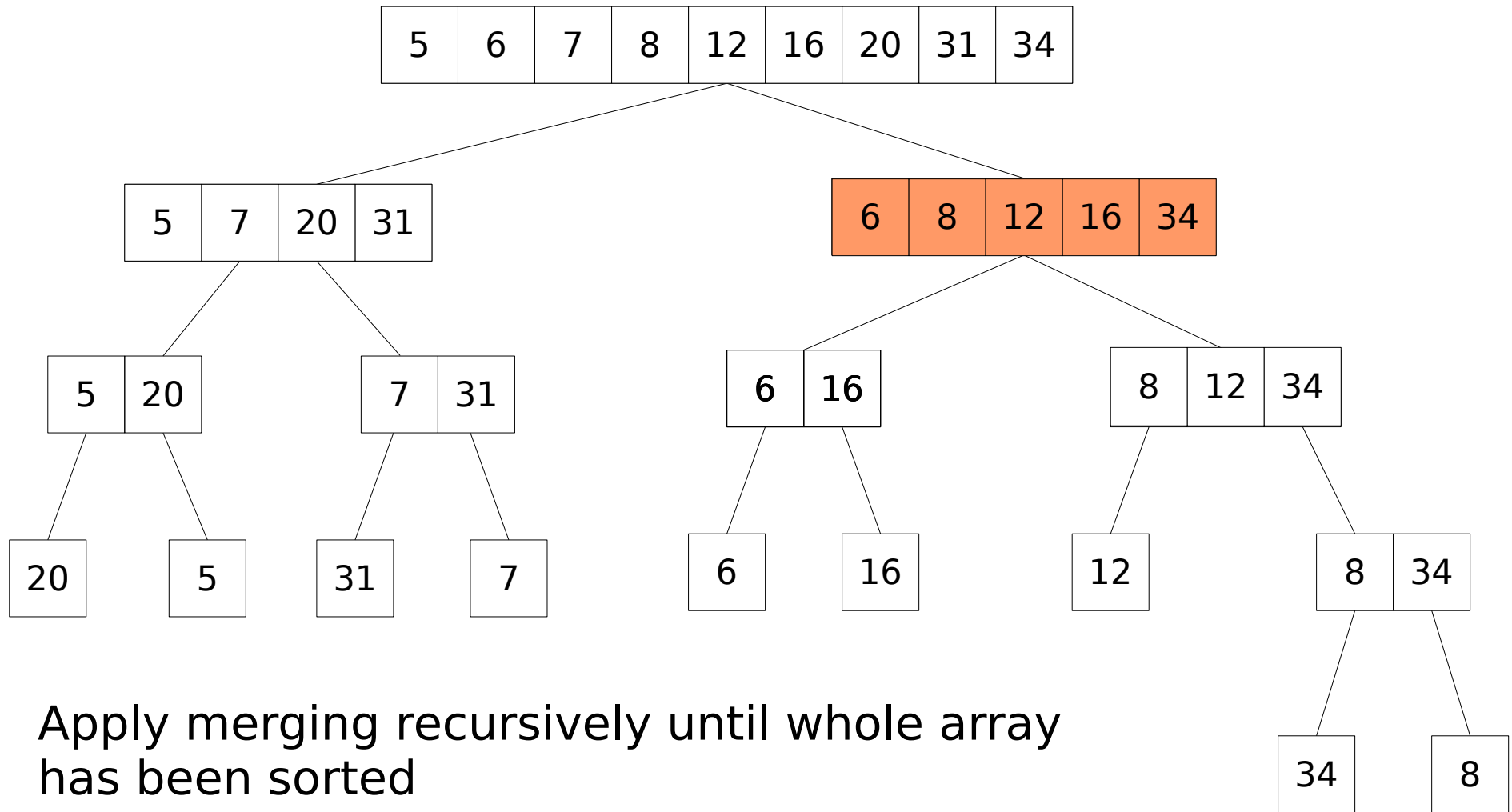
Merge Sort Visualization



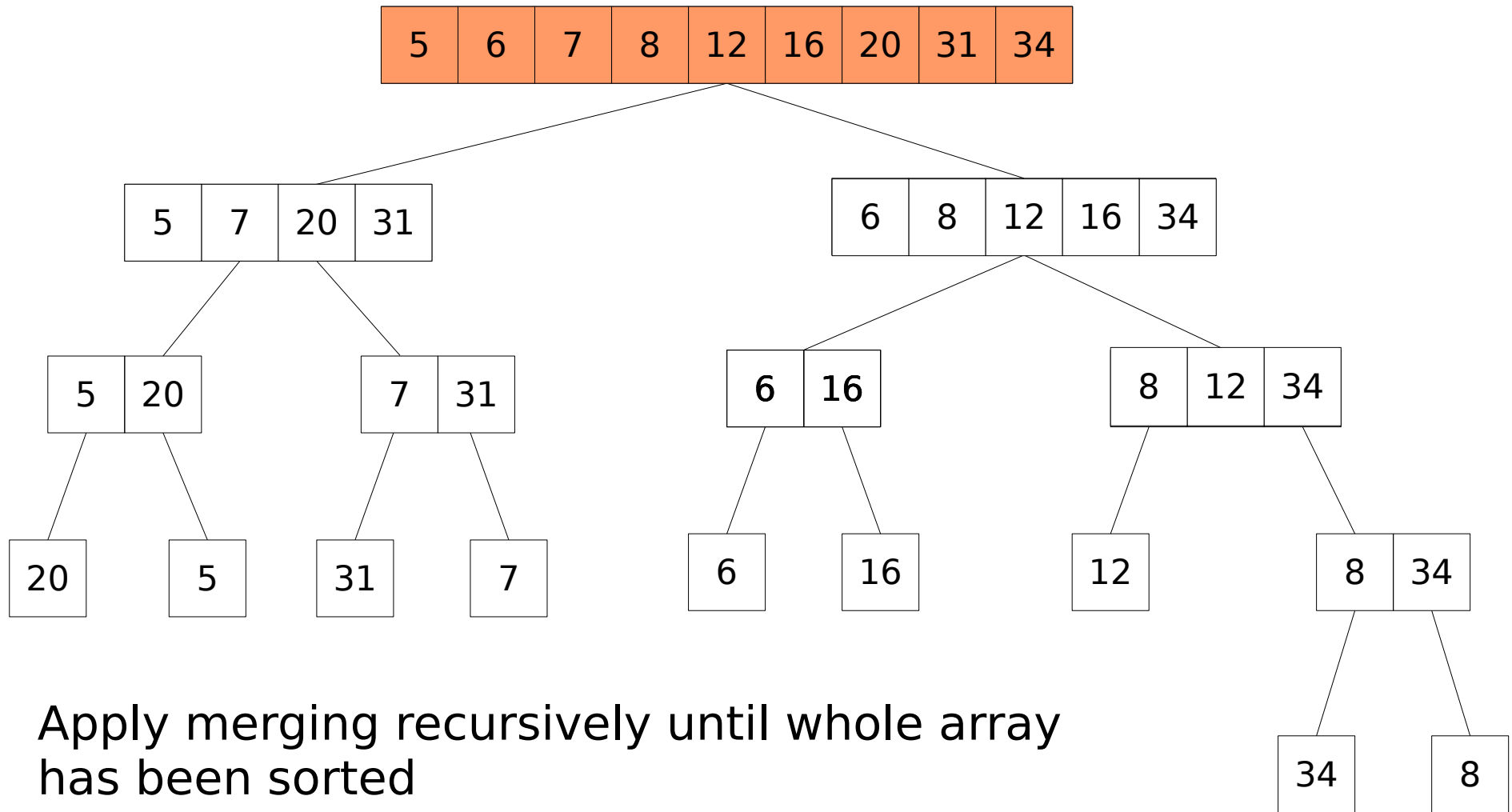
Merge Sort Visualization



Merge Sort Visualization



Merge Sort Visualization



- Apply merging recursively until whole array has been sorted
- See [MergeSort.java](#) under SavitchSrc on the course website