

4.17 Recursive Searching and Sorting

Self-Study Guide · About 90 minutes · With answer key

By the end you will be able to

- Trace recursive methods that work through a String, an array, or an ArrayList one position at a time.
- Explain why binary search needs sorted data, and trace each step of a binary search, written with a loop or with recursion.
- Compare how many elements binary search and linear search check, and explain why binary search is usually faster.
- Trace merge sort as it splits an array into one-element pieces and merges them back together in sorted order.

What to remember from earlier

- A recursive method calls itself and needs a base case to stop. ([Topic 4.16: Recursion, 4.16.2 The Base Case Stops the Recursion](#))
- Every call has its own copy of the parameters and local variables. ([Topic 4.16: Recursion, 4.16.3 Each Call Has Its Own Variables](#))
- Linear search checks elements one at a time until it finds the target or runs out. ([Topic 4.14: Searching Algorithms, 4.14.1 Checking Each Element in Order](#))
- Selection sort and insertion sort put an array in order using loops. ([Topic 4.15: Sorting Algorithms](#))
- `substring(1)` gives every character of a String except the first. ([Topic 1.15: String Manipulation, 1.15.5 Extracting Text with substring](#))

4.17.1 Recursion on Strings

In Topic 4.16 you traced recursive methods that worked on numbers: a method that counts down from `n` calls itself with `n - 1` until it reaches a base case such as `n == 0`. Each call handles one small piece of the job and trusts the recursive call to handle the rest.

A String can be broken into pieces the same way. Any word is "its first letter" followed by "the rest of the word", and the rest of the word is a shorter String. In Topic 1.15 you met the two tools that split a word this way:

Expression	Gives	For "cat"
<code>word.substring(0, 1)</code>	the first character only	"c"
<code>word.substring(1)</code>	everything after the first	"at"

So a recursive method can look at the first letter, then call itself on `word.substring(1)`. Each call gets a word one character shorter, so the calls must eventually reach the empty string `""`, which has length 0. That makes a natural base case: there is nothing left to look at. Working through a String, an array, or an ArrayList this way, one position per call, is called a **recursive traversal**. It

does the same job as a loop that visits every position (Topic 4.16 showed that anything a loop can repeat, recursion can repeat too).

KEY IDEA

A recursive traversal of a String handles one character per call and passes a shorter String, usually `word.substring(1)`, to the recursive call. The base case is a String too short to split, usually length 0 or 1.

WORKED EXAMPLE Counting one letter in a word

`countLetter` returns how many times `letter` appears in `word`. Look for three things: the base case on line 5, the recursive call on line 9 that passes a shorter word, and line 10, which looks at the first letter **after** the recursive call has returned.

CountLetter.java

```

1 public class CountLetter
2 {
3     public static int countLetter(String word, String letter)
4     {
5         if (word.length() == 0)
6         {
7             return 0;
8         }
9         int restCount = countLetter(word.substring(1), letter);
10        if (word.substring(0, 1).equals(letter))
11        {
12            return 1 + restCount;
13        }
14        return restCount;
15    }
16
17    public static void main(String[] args)
18    {
19        System.out.println(countLetter("banana", "a"));
20        System.out.println(countLetter("moon", "o"));
21    }
22 }
```

The call `countLetter("banana", "a")` cannot finish until it knows how many `a`s are in `"anana"`, so it calls itself and waits. That call waits for `"nana"`, and so on, down to `"`. The first table shows the calls going down:

Call	word	What happens
1	"banana"	calls <code>countLetter("anana", "a")</code> , waits
2	"anana"	calls <code>countLetter("nana", "a")</code> , waits
3	"nana"	calls <code>countLetter("ana", "a")</code> , waits
4	"ana"	calls <code>countLetter("na", "a")</code> , waits
5	"na"	calls <code>countLetter("a", "a")</code> , waits
6	"a"	calls <code>countLetter("", "a")</code> , waits

7	""	base case: returns 0
---	----	----------------------

Now the answers come back up, newest call first. Each waiting call gets `restCount`, checks its own first letter on line 10, and returns:

Call	word	restCount	First letter is "a" ?	Returns
6	"a"	0	yes	1
5	"na"	1	no	1
4	"ana"	1	yes	2
3	"nana"	2	no	2
2	"anana"	2	yes	3
1	"banana"	3	no	3

3
2

The second line comes from `countLetter("moon", "o")`: the calls go "moon", "oon", "on", "n", "", and the two o's add 1 each on the way back up.

The next example changes one thing: instead of returning a count, the method **builds a new String** on the way back up. Where the method puts the first letter decides the result.

WORKED EXAMPLE Reversing a word

ReverseWord.java

```

1 public class ReverseWord
2 {
3     public static String reverse(String word)
4     {
5         if (word.length() <= 1)
6         {
7             return word;
8         }
9         return reverse(word.substring(1)) + word.substring(0, 1);
10    }
11
12    public static void main(String[] args)
13    {
14        System.out.println(reverse("cat"));
15        System.out.println(reverse("stop"));
16    }
17 }
```

The base case on line 5 stops at a word of length 0 or 1, since such a word reads the same backward. Line 9 reverses the rest of the word first, then glues the first letter on the **end**. Trace `reverse("cat")`:

Call	word	Waits for	Returns
3	"t"	nothing; base case	"t"

2	"at"	reverse("t")	"t" + "a" = "ta"
1	"cat"	reverse("at")	"ta" + "c" = "tac"

tac
pots

Read the table from the bottom row up to see the order the calls start, and from the top row down to see the order they return. The first letter of each call is added last, which is exactly what turns the word around. If line 9 were `word.substring(0, 1) + reverse(word.substring(1))`, every letter would go back in its original place and the method would return the word unchanged.

WATCH OUT

Forgetting to make the String shorter. A student writes the recursive call with `word` instead of `word.substring(1)`:

CountLetterNoProgress.java

```

1 public class CountLetterNoProgress
2 {
3     public static int countLetter(String word, String letter)
4     {
5         if (word.length() == 0)
6         {
7             return 0;
8         }
9         int restCount = countLetter(word, letter);
10        if (word.substring(0, 1).equals(letter))
11        {
12            return 1 + restCount;
13        }
14        return restCount;
15    }
16
17    public static void main(String[] args)
18    {
19        System.out.println(countLetter("banana", "a"));
20    }
21 }
```

Every call passes "banana" again, so the base case `word.length() == 0` is never reached. As in Topic 4.16, each waiting call stays on the call stack, which has limited room. After thousands of calls it runs out, and the program stops with an error:

```

Exception in thread "main" java.lang.StackOverflowError
  at CountLetterNoProgress.countLetter(CountLetterNoProgress.java:9)
  ...
```

A `StackOverflowError` almost always means the recursion never reaches its base case. Check that every recursive call passes something smaller (here, `word.substring(1)`), so the calls move toward the base case.

PAUSE AND TRY

Question 1 Tracing

Trace this program and write its exact output. For `everyOther("planet")`, list each call's `text` and what it returns.

EveryOther.java

```
1 public class EveryOther
2 {
3     public static String everyOther(String text)
4     {
5         if (text.length() <= 1)
6         {
7             return text;
8         }
9         return text.substring(0, 1) + everyOther(text.substring(2));
10    }
11
12    public static void main(String[] args)
13    {
14        System.out.println(everyOther("planet"));
15        System.out.println(everyOther("robot"));
16    }
17 }
```

Question 2 Multiple choice

Using `countLetter` from `CountLetter.java`, how many times in total is `countLetter` called when `main` runs `countLetter("moon", "o")`? Count the first call too.

- A. 2
- B. 4
- C. 5
- D. 6

4.17.2 Recursion on Arrays and ArrayLists

You cannot chop the first element off an array the way `substring(1)` chops a letter off a `String`; an array's length is fixed when it is created (Topic 4.3). So a recursive method on an array keeps the whole array and adds a second parameter: the **index** of the element this call should handle. Each recursive call passes `index + 1`. The index plays the same role as the loop control variable in a `for` loop that traverses an array (Topic 4.4): it records how far the traversal has gone. Topic 4.16 described this idea as parameters capturing the progress of the recursion.

The base case is the index that is one past the last element, `nums.length`, because then there is nothing left to handle.

KEY IDEA

A recursive traversal of an array or `ArrayList` passes the same collection to every call along with an index. Each call handles the element at `index` and calls itself with `index + 1`; the base case is reached when the index equals the length (or size).

WORKED EXAMPLE Adding up an array recursively

SumFrom.java

```

1 public class SumFrom
2 {
3     public static int sumFrom(int[] nums, int index)
4     {
5         if (index == nums.length)
6         {
7             return 0;
8         }
9         return nums[index] + sumFrom(nums, index + 1);
10    }
11
12    public static void main(String[] args)
13    {
14        int[] points = {4, 7, 1, 8};
15        System.out.println(sumFrom(points, 0));
16        System.out.println(sumFrom(points, 2));
17    }
18 }
```

`sumFrom(nums, index)` returns the total of the elements from `index` to the end. Line 9 says: "my element, plus the total of everything after me." Trace `sumFrom(points, 0)` with `points = {4, 7, 1, 8}`:

Call	index	Line 9 waits for	Returns
5	4	nothing; base case	0
4	3	<code>sumFrom(points, 4) = 0</code>	<code>8 + 0 = 8</code>
3	2	<code>sumFrom(points, 3) = 8</code>	<code>1 + 8 = 9</code>
2	1	<code>sumFrom(points, 2) = 9</code>	<code>7 + 9 = 16</code>
1	0	<code>sumFrom(points, 1) = 16</code>	<code>4 + 16 = 20</code>

20
9

The second call, `sumFrom(points, 2)`, starts partway through, so it adds only `1 + 8`. The starting index you pass decides where the traversal begins. All five calls share one array; only `index` is different in each call.

The same pattern works for an `ArrayList`, using `size()` and `get` from Topic 4.8. This next method is a linear search (Topic 4.14) written recursively, and it has **two** base cases: one for "found it" and one for "ran out of elements".

WORKED EXAMPLE A recursive linear search on an ArrayList

FindName.java

```

1  import java.util.ArrayList;
2
3  public class FindName
4  {
5      public static int findFrom(ArrayList<String> names, String target, int index)
6      {
7          if (index == names.size())
8          {
9              return -1;
10         }
11         if (names.get(index).equals(target))
12         {
13             return index;
14         }
15         return findFrom(names, target, index + 1);
16     }
17
18     public static void main(String[] args)
19     {
20         ArrayList<String> team = new ArrayList<String>();
21         team.add("Ana");
22         team.add("Ravi");
23         team.add("Kai");
24         team.add("Mo");
25         System.out.println(findFrom(team, "Kai", 0));
26         System.out.println(findFrom(team, "Zoe", 0));
27     }
28 }
```

Line	Purpose
7	Base case 1: the index has passed the last element, so the name is absent; return -1
11	Base case 2: the element at <code>index</code> matches; return that index
15	Otherwise, search the rest of the list starting at <code>index + 1</code>

For `findFrom(team, "Kai", 0)`: index 0 holds "Ana", no match; index 1 holds "Ravi", no match; index 2 holds "Kai", so that call returns 2. Each waiting call returns that 2 unchanged, because line 15 returns whatever the recursive call returns. For "Zoe", the calls run through indexes 0, 1, 2, and 3 without a match, and the call with index 4 hits the first base case and returns -1.

2
-1

WATCH OUT

A base case that stops one step too late. The base case must stop at `index == nums.length`, because `nums.length` is not a valid index. A student writes `>` instead:

SumFromPastEnd.java

```
1 public class SumFromPastEnd
2 {
3     public static int sumFrom(int[] nums, int index)
4     {
5         if (index > nums.length)
6         {
7             return 0;
8         }
9         return nums[index] + sumFrom(nums, index + 1);
10    }
11
12    public static void main(String[] args)
13    {
14        int[] points = {4, 7, 1, 8};
15        System.out.println(sumFrom(points, 0));
16    }
17 }
```

When `index` is 4, the test `4 > 4` is `false`, so the method does not stop. Line 9 then reads `nums[4]` in an array whose last index is 3, and Java throws an exception:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of
bounds for length 4
```

(The exact wording can vary between Java versions.) This is the same off-by-one mistake as writing `i <= nums.length` in a `for` loop. The fix is `if (index == nums.length)`, or equivalently `index >= nums.length`.

PAUSE AND TRY**Question 3** Tracing

This method prints **after** its recursive call instead of before. Write the exact output.

PrintBackward.java

```
1 public class PrintBackward
2 {
3     public static void printBackward(int[] nums, int index)
4     {
5         if (index < nums.length)
6         {
7             printBackward(nums, index + 1);
8             System.out.println(nums[index]);
9         }
10    }
11
12    public static void main(String[] args)
13    {
14        int[] laps = {12, 30, 5};
15        printBackward(laps, 0);
16    }
17 }
```

Question 4 Multiple choice

Using `sumFrom` from `SumFrom.java`, `prices` is `{6, 2, 9, 5}`. Which call returns 22, the total of all four prices?

- A. `sumFrom(prices, 0)`
- B. `sumFrom(prices, 1)`
- C. `sumFrom(prices, prices.length)`
- D. `sumFrom(prices, prices.length - 1)`

4.17.3 The Binary Search Idea

Play a guessing game: a friend picks a whole number from 1 to 100 and answers each guess with "higher", "lower", or "got it". Guessing 1, 2, 3, ... in order is a linear search (Topic 4.14), and it could take 100 guesses. A smarter player guesses 50 first. Whatever the answer, half of the numbers are gone in one guess. Then they guess the middle of what is left, and so on.

That strategy works only because the numbers are **in order**. "Higher" tells you the secret number is above 50 because every number above 50 is bigger. Searching an array this way is called **binary search**: look at the middle element of the part still being searched, compare it with the target, and throw away the half that cannot contain the target. ("Binary" refers to the choice between two halves.) Binary search repeats until it finds the target or nothing is left.

The data must be sorted for this to work. In Topic 1.8 you met the word precondition: a condition that must be true before code runs, or the code cannot be trusted to work. Sorted order is binary search's precondition. In this chapter, "sorted" means smallest to largest.

To keep track of the part still being searched, binary search uses two indexes: `low` (the first index still possible) and `high` (the last). The middle index is `mid = (low + high) / 2`, computed with `int` division, so it rounds down when `low + high` is odd.

KEY IDEA

Binary search works only on sorted data. It checks the middle element of the range from `low` to `high`; if that element is too small, the target can only be to the right, and if it is too large, the target can only be to the left. Each check eliminates half of the remaining elements.

WORKED EXAMPLE Searching for 26 by hand

The sorted array `ages` has 9 elements. Search for 26:

```
index:  0   1   2   3   4   5   6   7   8
value:  3   8  12  17  21  26  30  34  41
```

Step	low	high	mid = (low + high) / 2	ages[mid]	Decision
1	0	8	4	21	21 is less than 26: go right, low = 5
2	5	8	6 (13 / 2 rounds down)	30	30 is more than 26: go left, high = 5
3	5	5	5	26	found at index 5

Here is what is left after each step (dots mark eliminated elements):

```
start:    3   8  12  17  21  26  30  34  41
step 1:   .   .   .   .   .  26  30  34  41
step 2:   .   .   .   .   .  26   .   .   .
step 3:  found 26 at index 5
```

Notice that `mid` itself is eliminated after each check. Once 21 is known to be too small, there is no reason to look at it again, so `low` becomes `mid + 1`, not `mid`. Three checks found the target; a linear search would have needed six.

What if the target is not in the array? Binary search keeps shrinking the range until there is nothing left. You can tell nothing is left when `low` passes `high`.

WORKED EXAMPLE Searching for 10, which is not there

Step	low	high	mid	ages[mid]	Decision
1	0	8	4	21	21 is more than 10: high = 3
2	0	3	1	8	8 is less than 10: low = 2
3	2	3	2	12	12 is more than 10: high = 1
4	2	1			low is past high: nothing left; not found

The last range would run from index 2 to index 1, which contains no elements. That is binary search's signal that the target is absent. The value 10 would belong between 8 and 12, exactly where the search ended up.

WATCH OUT

Running binary search on unsorted data. Binary search does not check whether the data is sorted; it simply trusts it. Here is the array `{40, 12, 33, 7, 25, 18, 50}`, searched for 12, which is at index 1:

Step	low	high	mid	Value	Decision
1	0	6	3	7	7 is less than 12: go right
2	4	6	5	18	18 is more than 12: go left
3	4	4	4	25	25 is more than 12: go left
4	4	3			nothing left: reports not found

The search threw away the left half after step 1, on the assumption that everything left of 7 is smaller than 7. In this array that is false: 12 sits there. Binary search gives a wrong answer, with no error message. Sort the data first (merge sort, in 4.17.7, is one way), or use linear search, which works on any order.

PAUSE AND TRY**Question 5** Tracing

The sorted array `nums` is `{4, 11, 15, 23, 29, 36, 42}`. Using the steps from this subsection, list the values that binary search compares with the target, in order, and the result, when the target is (a) 36 and (b) 5.

Question 6 Multiple choice

Binary search, as described in this subsection, is guaranteed to give a correct answer on which array?

- A. `{5, 12, 9, 20, 31}`
- B. `{2, 7, 7, 15, 30}`
- C. `{40, 10, 25, 35, 50}`
- D. `{8, 6, 4, 9, 12}`

4.17.4 Binary Search with a Loop

Now turn the hand steps into Java. Each step of the hand trace did the same thing with different `low` and `high` values, and repeating the same steps while a condition holds is what a `while` loop is for (Topic 2.7). The condition is "there is still something left to search", which is `low <= high`. As in Topic 4.16, a version written with a loop is called iterative.

KEY IDEA

Iterative binary search repeats while `low <= high`: compute `mid`, return `mid` if `nums[mid]` is the target, otherwise move `low` to `mid + 1` or `high` to `mid - 1`. If the loop ends without finding the target, return -1.

WORKED EXAMPLE Iterative binary search with a trace line

Line 10 prints `low`, `high`, and `mid` at each pass so you can watch the range shrink. A real search would leave that line out; it does not change what the method returns.

BinarySearchLoop.java

```

1  public class BinarySearchLoop
2  {
3      public static int binarySearch(int[] nums, int target)
4      {
5          int low = 0;
6          int high = nums.length - 1;
7          while (low <= high)
8          {
9              int mid = (low + high) / 2;
10             System.out.println("low " + low + ", high " + high + ", mid " + mid);
11             if (nums[mid] == target)
12             {
13                 return mid;
14             }
15             else if (nums[mid] < target)
16             {
17                 low = mid + 1;
18             }
19             else
20             {
21                 high = mid - 1;
22             }
23         }
24         return -1;
25     }
26
27     public static void main(String[] args)
28     {
29         int[] ages = {3, 8, 12, 17, 21, 26, 30, 34, 41};
30         System.out.println("Index of 26: " + binarySearch(ages, 26));
31         System.out.println("Index of 10: " + binarySearch(ages, 10));
32     }
33 }

```

```

low 0, high 8, mid 4
low 5, high 8, mid 6
low 5, high 5, mid 5
Index of 26: 5
low 0, high 8, mid 4
low 0, high 3, mid 1
low 2, high 3, mid 2
Index of 10: -1

```

Match each piece of code to the hand trace in 4.17.3:

Lines	What it does
5–6	Start with the whole array: <code>low = 0</code> , <code>high = last index</code>

7	Keep going while the range from low to high holds at least one element
9	Find the middle index (int division rounds down)
11–14	Found it: return the index right away, which also ends the loop
15–18	Middle value too small: the target can only be to the right
19–22	Middle value too large: the target can only be to the left
24	Loop ended because low passed high: the target is absent

The printed lines match both hand traces exactly. In the search for 10, the third pass sets high to 1, the loop test `2 <= 1` is false, and line 24 returns -1. Returning -1 for "not found" is the same convention as linear search in Topic 4.14, chosen because -1 can never be a real index.

The same algorithm works on an `ArrayList`, and on Strings. Only the way you read an element and compare it changes:

In the <code>int[]</code> version	In an <code>ArrayList<String></code> version
<code>nums.length - 1</code>	<code>names.size() - 1</code>
<code>nums[mid]</code>	<code>names.get(mid)</code>
<code>nums[mid] == target</code>	<code>target.compareTo(names.get(mid)) == 0</code>
<code>nums[mid] < target</code>	<code>target.compareTo(names.get(mid)) > 0</code> (target comes later)

Remember from Topic 1.15 that `a.compareTo(b)` is positive when `a` comes after `b` in alphabetical order. For Strings, "sorted" means in alphabetical order.

WATCH OUT

Stopping when low equals high. A student writes `while (low < high)` on line 7. It looks harmless, but it skips the last check:

BinarySearchLessThan.java

```

1 public class BinarySearchLessThan
2 {
3     public static int binarySearch(int[] nums, int target)
4     {
5         int low = 0;
6         int high = nums.length - 1;
7         while (low < high)
8         {
9             int mid = (low + high) / 2;
10            if (nums[mid] == target)
11            {
12                return mid;
13            }
14            else if (nums[mid] < target)
15            {
16                low = mid + 1;
17            }
18            else
19            {
20                high = mid - 1;

```

```

21         }
22     }
23     return -1;
24 }
25
26 public static void main(String[] args)
27 {
28     int[] ages = {3, 8, 12, 17, 21, 26, 30, 34, 41};
29     System.out.println(binarySearch(ages, 21));
30     System.out.println(binarySearch(ages, 26));
31 }
32 }

```

```

4
-1

```

Searching for 21 works, because 21 is found in the first pass. Searching for 26 reaches `low = 5` and `high = 5`, a range with exactly one element, the 26. The test `5 < 5` is `false`, so the loop ends without checking it and the method returns `-1` even though 26 is in the array. The condition must be `low <= high`, because a range where `low` equals `high` still holds one element.

PAUSE AND TRY

Question 7 Tracing

Write the exact output of this program.

LoopSearchPractice.java

```

1 public class LoopSearchPractice
2 {
3     public static int binarySearch(int[] nums, int target)
4     {
5         int low = 0;
6         int high = nums.length - 1;
7         while (low <= high)
8         {
9             int mid = (low + high) / 2;
10            System.out.println(low + " " + high + " " + mid);
11            if (nums[mid] == target)
12            {
13                return mid;
14            }
15            else if (nums[mid] < target)
16            {
17                low = mid + 1;
18            }
19            else
20            {
21                high = mid - 1;
22            }

```

```
23     }
24     return -1;
25 }
26
27 public static void main(String[] args)
28 {
29     int[] times = {5, 9, 14, 20, 23, 31, 38, 44, 52, 60};
30     System.out.println(binarySearch(times, 52));
31     System.out.println(binarySearch(times, 4));
32 }
33 }
```

Question 8 Debugging

This search is supposed to print 3 and then 5, the indexes of 140 and 160. It prints 3 and then -1. Find the line with the error, explain why it gives the wrong result, and write the corrected line.

SwappedSearch.java

```
1 public class SwappedSearch
2 {
3     public static int binarySearch(int[] nums, int target)
4     {
5         int low = 0;
6         int high = nums.length - 1;
7         while (low <= high)
8         {
9             int mid = (low + high) / 2;
10            if (nums[mid] == target)
11            {
12                return mid;
13            }
14            else if (nums[mid] > target)
15            {
16                low = mid + 1;
17            }
18            else
19            {
20                high = mid - 1;
21            }
22        }
23        return -1;
24    }
25
26    public static void main(String[] args)
27    {
28        int[] heights = {110, 125, 132, 140, 151, 160, 174};
29        System.out.println(binarySearch(heights, 140));
30        System.out.println(binarySearch(heights, 160));
31    }
32 }
```

4.17.5 Binary Search with Recursion

Look again at what each pass of the loop did: it searched the range from `low` to `high` by checking the middle and then **searching a smaller range**. "Search a smaller range" is the same job on a smaller input, which is exactly what a recursive call does (Topic 4.16). So binary search can be written either iteratively, with a loop, or recursively, with the method calling itself.

In the recursive version, `low` and `high` become **parameters**. Each call searches its own range and either answers or calls itself with half the range. Because each call has its own copy of the parameters, each call remembers its own `low` and `high`.

KEY IDEA

Recursive binary search has two base cases: `low > high` (nothing left; return -1) and `nums[mid] == target` (return `mid`). Otherwise it returns the result of a recursive call on the left half (`low` to `mid - 1`) or the right half (`mid + 1` to `high`).

WORKED EXAMPLE Recursive binary search

Line 5 prints each call's parameters so you can see the calls. The first call, from `main`, passes the whole array: 0 and `ages.length - 1`.

BinarySearchRecursive.java

```
1 public class BinarySearchRecursive
2 {
3     public static int binarySearch(int[] nums, int target, int low, int high)
4     {
5         System.out.println("call: low " + low + ", high " + high);
6         if (low > high)
7         {
8             return -1;
9         }
10        int mid = (low + high) / 2;
11        if (nums[mid] == target)
12        {
13            return mid;
14        }
15        else if (nums[mid] < target)
16        {
17            return binarySearch(nums, target, mid + 1, high);
18        }
19        else
20        {
21            return binarySearch(nums, target, low, mid - 1);
22        }
23    }
24
25    public static void main(String[] args)
26    {
27        int[] ages = {3, 8, 12, 17, 21, 26, 30, 34, 41};
28        System.out.println("Index of 26: " + binarySearch(ages, 26, 0, ages.length
- 1));
```

```

29         System.out.println("Index of 10: " + binarySearch(ages, 10, 0, ages.length
30             - 1));
31     }
32 }

```

```

call: low 0, high 8
call: low 5, high 8
call: low 5, high 5
Index of 26: 5
call: low 0, high 8
call: low 0, high 3
call: low 2, high 3
call: low 2, high 1
Index of 10: -1

```

Trace the search for 26:

Call	low	high	mid	ages[mid]	What the call does
1	0	8	4	21	too small: returns the result of call 2 (line 17)
2	5	8	6	30	too large: returns the result of call 3 (line 21)
3	5	5	5	26	base case: returns 5 (line 13)

Call 3 returns 5 to call 2, which returns that 5 to call 1, which returns it to `main`. Nothing is added or changed on the way back; each call simply passes the answer up. That is why lines 17 and 21 say `return binarySearch(...)`.

For 10, the fourth call receives `low = 2` and `high = 1`. Line 6 sees that `low > high` and returns -1. Compare with the loop: there, `low` passing `high` made the loop condition false; here, it triggers the base case.

The two versions check the same elements in the same order. Only the bookkeeping differs:

Job	Iterative (<code>BinarySearchLoop</code>)	Recursive (<code>BinarySearchRecursive</code>)
Start with the whole array	<code>low = 0; high = nums.length - 1;</code>	First call passes 0 and <code>nums.length - 1</code>
Stop when nothing is left	Loop condition <code>low <= high</code> becomes false	Base case <code>low > high</code> returns -1
Stop when found	<code>return mid;</code> inside the loop	Base case <code>return mid;</code>
Move to the right half	<code>low = mid + 1;</code> then loop again	<code>return binarySearch(nums, target, mid + 1, high);</code>
Move to the left half	<code>high = mid - 1;</code> then loop again	<code>return binarySearch(nums, target, low, mid - 1);</code>

WATCH OUT

A recursive call that does not shrink the range. A student writes `mid` instead of `mid + 1` on line 16:

RecursiveNoProgress.java

```
1 public class RecursiveNoProgress
2 {
3     public static int binarySearch(int[] nums, int target, int low, int high)
4     {
5         if (low > high)
6         {
7             return -1;
8         }
9         int mid = (low + high) / 2;
10        if (nums[mid] == target)
11        {
12            return mid;
13        }
14        else if (nums[mid] < target)
15        {
16            return binarySearch(nums, target, mid, high);
17        }
18        else
19        {
20            return binarySearch(nums, target, low, mid - 1);
21        }
22    }
23
24    public static void main(String[] args)
25    {
26        int[] ages = {3, 8, 12, 17, 21, 26, 30, 34, 41};
27        System.out.println(binarySearch(ages, 30, 0, ages.length - 1));
28        System.out.println(binarySearch(ages, 41, 0, ages.length - 1));
29    }
30 }
```

6

Exception in thread "main" java.lang.StackOverflowError

Searching for 30 works, by luck. Searching for 41 does not: the calls narrow to `low = 7` and `high = 8`, so `mid = 7` and `ages[7] = 34` is too small. The next call gets `low = mid = 7` and `high = 8` again, exactly the same range. Every later call is identical, so neither base case is ever reached and the call stack runs out of room. Using `mid + 1` (and `mid - 1` on the other side) guarantees that each call's range is at least one element smaller than the last, because the middle element has already been checked.

PAUSE AND TRY

Question 9 Multiple choice

Using `BinarySearchRecursive.java`, a program searches `ages` for 34 with `binarySearch(ages, 34, 0, ages.length - 1)`. What are `low` and `high` in the **second** call, and how many calls are made in total?

A. `low 5, high 8`; 3 calls

- B. low 4, high 8; 3 calls
- C. low 5, high 8; 4 calls
- D. low 0, high 4; 2 calls

Question 10 Short answer

In the iterative version, the `while` loop stops when `low <= high` becomes false. What plays that role in the recursive version, and why does each recursive call need the word `return` in front of it?

4.17.6 Comparing Binary Search and Linear Search

Binary search needs sorted data and more careful code than linear search. Is it worth it? In Topic 2.12 you compared algorithms informally by counting how many times a statement runs. Here the statement worth counting is "check one element against the target".

Think about the worst case for each. Linear search on 1000 elements may check all 1000. Binary search throws away half of what is left with every check, so the number of elements still possible goes:

1000 → 500 → 250 → 125 → 62 → 31 → 15 → 7 → 3 → 1

That is nine halvings to get down to one element, plus one check of that last element: at most 10 checks. Doubling the array to 2000 elements adds only one more halving, so only one more check. Linear search's worst case doubles.

KEY IDEA

Binary search usually needs far fewer checks than linear search: each check eliminates half of the remaining elements, so doubling the amount of data adds only one more check in the worst case, while linear search's worst case doubles.

WORKED EXAMPLE Counting checks on 1000 sorted ids

This program fills an array with 1000 even numbers (0, 2, 4, ..., 1998), then counts the checks each search makes for four targets. The two methods are the searches you already know, each with a counter added.

SearchCompare.java

```
1 public class SearchCompare
2 {
3     public static int linearChecks(int[] nums, int target)
4     {
5         int checks = 0;
6         for (int i = 0; i < nums.length; i++)
7         {
8             checks++;
9             if (nums[i] == target)
10            {
```

```

11         return checks;
12     }
13 }
14 return checks;
15 }
16
17 public static int binaryChecks(int[] nums, int target)
18 {
19     int checks = 0;
20     int low = 0;
21     int high = nums.length - 1;
22     while (low <= high)
23     {
24         int mid = (low + high) / 2;
25         checks++;
26         if (nums[mid] == target)
27         {
28             return checks;
29         }
30         else if (nums[mid] < target)
31         {
32             low = mid + 1;
33         }
34         else
35         {
36             high = mid - 1;
37         }
38     }
39     return checks;
40 }
41
42 public static void main(String[] args)
43 {
44     int[] ids = new int[1000];
45     for (int i = 0; i < ids.length; i++)
46     {
47         ids[i] = 2 * i;
48     }
49     int[] targets = {0, 1000, 1998, 1999};
50     for (int target : targets)
51     {
52         System.out.println(target + ": linear " + linearChecks(ids, target)
53             + ", binary " + binaryChecks(ids, target));
54     }
55 }
56 }

```

```

0: linear 1, binary 9
1000: linear 501, binary 9
1998: linear 1000, binary 10
1999: linear 1000, binary 10

```

Target	Where it is	Linear checks	Binary checks
--------	-------------	---------------	---------------

0	index 0 (very first)	1	9
1000	index 500 (middle)	501	9
1998	index 999 (very last)	1000	10
1999	not in the array	1000	10

Binary search never needed more than 10 checks. Linear search needed up to 1000.

The gap grows as the data grows. Searching for a value larger than every element is one of binary search's worst cases; counting the checks for that search on bigger arrays gives:

Number of elements	Linear search, worst case	Binary search, worst case
10	10	4
100	100	7
1000	1000	10
1,000,000	1,000,000	20

WATCH OUT

"Binary search is always faster." Look at the first row of the output above. The target 0 was the very first element, so linear search found it in 1 check, while binary search took 9, because it starts in the middle and works inward. That is why binary search is described as **typically** more efficient, not always. It is also why linear search still matters: it is the right tool when the data is not sorted, since binary search would give wrong answers there (4.17.3).

PAUSE AND TRY

Question 11 Multiple choice

A sorted array holds 100 elements. What is the greatest number of elements binary search could check before it finds the target or decides it is absent?

- A. 7
- B. 10
- C. 50
- D. 100

Question 12 Short answer

A program searches a sorted array of 1000 ids, and almost every search is for one of the first two or three ids. A student says binary search is still the better choice "because binary search is always faster". Explain what is wrong with the student's reasoning, using the counts from `SearchCompare.java`.

4.17.7 Merge Sort: Split, Then Merge

Binary search needs sorted data, and Topic 4.15 gave you two ways to sort: selection sort and insertion sort. Both are iterative; they use nested loops. This subsection introduces a third sorting algorithm that is **recursive**, called **merge sort**. It rests on two observations:

1. An array with only one element is already sorted. There is nothing to put in order.
2. If you have two piles that are each already sorted, combining them into one sorted pile is easy: keep taking the smaller of the two top cards.

So merge sort does not try to sort a big array directly. It splits the array in half, splits each half in half, and keeps going until every piece, or **subarray**, holds a single element. Then it **merges** the pieces back together, two at a time, each merge producing a longer sorted piece, until one sorted array remains.

The split rule used in this chapter: the left piece gets $\text{length} / 2$ elements (with `int` division), and the right piece gets the rest. When the length is odd, the right piece gets the extra element.

KEY IDEA

Merge sort splits an array into halves, again and again, until every subarray has one element. Then it merges sorted subarrays back together in pairs, keeping the values in order, until the whole array is one sorted piece.

WORKED EXAMPLE Merge sort on 8 scores

Sort {52, 17, 38, 9, 61, 24, 45, 30}. The top half of the diagram splits; the bottom half merges.

```

                    [52, 17, 38, 9, 61, 24, 45, 30]
split:      [52, 17, 38, 9]      [61, 24, 45, 30]
split:      [52, 17] [38, 9]      [61, 24] [45, 30]
split:      [52] [17] [38] [9]      [61] [24] [45] [30]
merge:      [17, 52] [9, 38]      [24, 61] [30, 45]
merge:      [9, 17, 38, 52]      [24, 30, 45, 61]
merge:      [9, 17, 24, 30, 38, 45, 52, 61]

```

Step by step:

- **Splitting** (three levels): 8 elements become two pieces of 4, then four pieces of 2, then eight pieces of 1. No values are compared or moved past each other; each piece keeps the original order.
- **First merges**: each pair of one-element pieces becomes a sorted pair. [52] and [17] become [17, 52] because 17 is smaller.
- **Second merges**: [17, 52] and [9, 38] become [9, 17, 38, 52], and [24, 61] and [30, 45] become [24, 30, 45, 61].
- **Final merge**: the two sorted halves become the sorted array.

All the actual sorting work happens while merging.

The array does not have to have a length that halves evenly. With an odd length, the pieces are just uneven.

WORKED EXAMPLE Merge sort on 5 values

Sort {6, 2, 9, 4, 1}. The left piece gets $5 / 2 = 2$ elements, the right gets 3. The right piece then splits into 1 and 2, and the 2 splits into 1 and 1.

```

          [6, 2, 9, 4, 1]
split:    [6, 2]      [9, 4, 1]
split:    [6] [2]    [9]   [4, 1]
split:                [4] [1]
merge:    [2, 6]      [9]   [1, 4]
merge:    [2, 6]      [1, 4, 9]
merge:          [1, 2, 4, 6, 9]

```

The piece [9] reaches one element sooner than [4, 1], so it simply waits until [4, 1] has been split and merged into [1, 4]. Then [9] and [1, 4] merge into [1, 4, 9]. Pieces of different sizes merge the same way as equal ones.

WATCH OUT

Thinking the split puts small values on the left. Some students picture the split like a sorting step, with the smaller values going left. It does not. The split cuts the array at a position, so [52, 17, 38, 9] is just the first four elements as they were. In the 5-value example, the left piece [6, 2] holds two of the larger values. Only merging compares values, so if you are asked what a piece holds right after a split, copy the elements in their original order.

PAUSE AND TRY**Question 13** Tracing

Draw the split and merge diagram for merge sort on {40, 15, 33, 8, 27, 12}, using this subsection's split rule.

Question 14 Multiple choice

Which statement about merge sort is true?

- A. Each split moves the smaller values into the left subarray.
- B. Splitting stops at subarrays of one element, because a one-element subarray is already sorted.
- C. Merge sort first sorts each half with selection sort, then merges the halves.
- D. Merge sort works only when the array's length can be halved evenly every time, such as 8 or 16.

4.17.8 Merging Two Sorted Halves

The diagrams in 4.17.7 merged pieces "in order", but how does a program do that? Picture two sorted stacks of exam papers, lowest score on top. To combine them into one sorted stack, compare the two top papers, move the lower one to the new stack, and repeat. When one stack runs out, the rest of the other stack goes on as it is, since it is already in order.

In Java, three index variables track the positions:

Variable	Tracks
i	the next unused element of <code>left</code>
j	the next unused element of <code>right</code>
k	the next empty spot in <code>result</code> , where the next value goes

KEY IDEA

To merge two sorted arrays, repeatedly compare the next unused element of each and copy the smaller one into the result. When one array runs out, copy whatever is left in the other array; those values are already in order.

WORKED EXAMPLE Merging two sorted halves of 4

`merge` fills `result` from `left` and `right`. The `show` method only builds a printable String like `[1, 2, 3]`, the way an `ArrayList` prints (Topic 4.8).

MergeHalves.java

```
1 public class MergeHalves
2 {
3     public static void merge(int[] result, int[] left, int[] right)
4     {
5         int i = 0;
6         int j = 0;
7         int k = 0;
8         while (i < left.length && j < right.length)
9         {
10             if (left[i] <= right[j])
11             {
12                 result[k] = left[i];
13                 i++;
14             }
15             else
16             {
17                 result[k] = right[j];
18                 j++;
19             }
20             k++;
21         }
22         while (i < left.length)
23         {
24             result[k] = left[i];
25             i++;
26             k++;
27         }
28         while (j < right.length)
29         {
30             result[k] = right[j];
31             j++;
32             k++;
33         }
```

```

34     }
35
36     public static String show(int[] nums)
37     {
38         String text = "[";
39         for (int i = 0; i < nums.length; i++)
40         {
41             if (i > 0)
42             {
43                 text = text + ", ";
44             }
45             text = text + nums[i];
46         }
47         return text + "]";
48     }
49
50     public static void main(String[] args)
51     {
52         int[] left = {9, 17, 38, 52};
53         int[] right = {24, 30, 45, 61};
54         int[] result = new int[8];
55         merge(result, left, right);
56         System.out.println(show(result));
57
58         int[] small = {5, 12};
59         int[] large = {3, 8, 20, 25};
60         int[] combined = new int[6];
61         merge(combined, small, large);
62         System.out.println(show(combined));
63     }
64 }

```

The first loop (lines 8–21) runs while **both** arrays still have unused elements. Trace it for `left = {9, 17, 38, 52}` and `right = {24, 30, 45, 61}`:

k	Compare (<code>left[i]</code> vs <code>right[j]</code>)	Copied	i, j after
0	9 vs 24	9	1, 0
1	17 vs 24	17	2, 0
2	38 vs 24	24	2, 1
3	38 vs 30	30	2, 2
4	38 vs 45	38	3, 2
5	52 vs 45	45	3, 3
6	52 vs 61	52	4, 3

Now `i` is 4, equal to `left.length`, so the first loop stops. The loop on lines 22–27 does nothing (`left` is used up), and the loop on lines 28–33 copies the leftover 61 into `result[7]`.

```

[9, 17, 24, 30, 38, 45, 52, 61]
[3, 5, 8, 12, 20, 25]

```

The second merge in `main` varies two things: the arrays have different lengths, and one runs out early.

WORKED EXAMPLE One side runs out early

Merge `small = {5, 12}` with `large = {3, 8, 20, 25}`:

k	Compare	Copied	i, j after
0	5 vs 3	3	0, 1
1	5 vs 8	5	1, 1
2	12 vs 8	8	1, 2
3	12 vs 20	12	2, 2

`small` is used up after only four values have been placed. The first loop stops, and the loop on lines 28–33 copies 20 and 25 without comparing them, because they were already in order in `large`. The result is `[3, 5, 8, 12, 20, 25]`.

Line 10 uses `<=`, so when the two next values are equal, the one from `left` is copied first. Either choice would still produce a sorted result.

WATCH OUT

Forgetting the leftovers. A student keeps only the first loop, thinking it handles everything:

`MergeNoLeftovers.java`

```

1 public class MergeNoLeftovers
2 {
3     public static void merge(int[] result, int[] left, int[] right)
4     {
5         int i = 0;
6         int j = 0;
7         int k = 0;
8         while (i < left.length && j < right.length)
9         {
10            if (left[i] <= right[j])
11            {
12                result[k] = left[i];
13                i++;
14            }
15            else
16            {
17                result[k] = right[j];
18                j++;
19            }
20            k++;
21        }
22    }
23
24    public static String show(int[] nums)
25    {
26        String text = "[";
27        for (int i = 0; i < nums.length; i++)
28        {

```

```

29         if (i > 0)
30         {
31             text = text + ", ";
32         }
33         text = text + nums[i];
34     }
35     return text + "];";
36 }
37
38 public static void main(String[] args)
39 {
40     int[] left = {9, 17, 38, 52};
41     int[] right = {24, 30, 45, 61};
42     int[] result = new int[8];
43     merge(result, left, right);
44     System.out.println(show(result));
45
46     int[] small = {5, 12};
47     int[] large = {3, 8, 20, 25};
48     int[] combined = new int[6];
49     merge(combined, small, large);
50     System.out.println(show(combined));
51 }
52 }

```

```

[9, 17, 24, 30, 38, 45, 52, 0]
[3, 5, 8, 12, 0, 0]

```

The first loop stops as soon as **either** array runs out, so the values still waiting in the other array are never copied. Those spots in `result` keep the default value 0 that every new `int` array starts with (Topic 4.3). The two leftover loops are not optional; at least one of them has work to do in almost every merge.

PAUSE AND TRY

Question 15 Tracing

Using `merge` from `MergeHalves.java`, trace the merge of `left = {4, 10, 15}` and `right = {6, 7, 20}`. List each comparison, the value copied, and the final contents of `result`.

Question 16 Multiple choice

During a call to `merge`, `left` has length 2 and `right` has length 4. The first loop ends with `i = 2` and `j = 1`. What happens next?

- A. Nothing; the merge is complete.
- B. The loop on lines 22–27 copies the rest of `left`.
- C. The loop on lines 28–33 copies 3 values from `right`.
- D. The loop on lines 28–33 copies 4 values from `right`.

4.17.9 The Recursive mergeSort Method

You now have both halves of merge sort: the split-and-merge plan (4.17.7) and a `merge` method (4.17.8). The `mergeSort` method connects them with recursion. To sort an array, it copies the left part into a new array `left` and the right part into a new array `right`, sorts each one **by calling `mergeSort` on it**, and then merges the two sorted arrays back into the original array.

The base case is an array of length 1 (or 0): it is already sorted, so the method does nothing. That is why the whole method body sits inside `if (nums.length > 1)`. When the test is false, the method simply returns; no `else` is needed.

KEY IDEA

`mergeSort` splits the array into new `left` and `right` arrays, calls itself on each, and then merges them back into the original array. The base case, an array with fewer than two elements, does nothing because it is already sorted.

WORKED EXAMPLE Watching every merge

Line 20 prints each merge as it happens. The `merge` and `show` methods are the same as in `MergeHalves.java`.

`MergeSortDemo.java`

```
1 public class MergeSortDemo
2 {
3     public static void mergeSort(int[] nums)
4     {
5         if (nums.length > 1)
6         {
7             int[] left = new int[nums.length / 2];
8             int[] right = new int[nums.length - left.length];
9             for (int i = 0; i < left.length; i++)
10            {
11                left[i] = nums[i];
12            }
13            for (int i = 0; i < right.length; i++)
14            {
15                right[i] = nums[left.length + i];
16            }
17            mergeSort(left);
18            mergeSort(right);
19            merge(nums, left, right);
20            System.out.println("merged " + show(left) + " and " + show(right) + "
21            into " + show(nums));
22        }
23    }
24    public static void merge(int[] result, int[] left, int[] right)
25    {
26        int i = 0;
27        int j = 0;
28        int k = 0;
```

```
29     while (i < left.length && j < right.length)
30     {
31         if (left[i] <= right[j])
32         {
33             result[k] = left[i];
34             i++;
35         }
36         else
37         {
38             result[k] = right[j];
39             j++;
40         }
41         k++;
42     }
43     while (i < left.length)
44     {
45         result[k] = left[i];
46         i++;
47         k++;
48     }
49     while (j < right.length)
50     {
51         result[k] = right[j];
52         j++;
53         k++;
54     }
55 }
56
57 public static String show(int[] nums)
58 {
59     String text = "[";
60     for (int i = 0; i < nums.length; i++)
61     {
62         if (i > 0)
63         {
64             text = text + ", ";
65         }
66         text = text + nums[i];
67     }
68     return text + "]";
69 }
70
71 public static void main(String[] args)
72 {
73     int[] scores = {52, 17, 38, 9, 61, 24, 45, 30};
74     mergeSort(scores);
75     System.out.println(show(scores));
76 }
77 }
```

Lines	What it does
5	Base case test: only arrays with 2 or more elements need work
7-8	Make the two new arrays; left gets length / 2 elements, right the rest

9–16	Copy the first part of <code>nums</code> into <code>left</code> and the rest into <code>right</code>
17–18	Sort each part recursively
19	Merge the two sorted parts back into <code>nums</code> , overwriting its old order

```
merged [52] and [17] into [17, 52]
merged [38] and [9] into [9, 38]
merged [17, 52] and [9, 38] into [9, 17, 38, 52]
merged [61] and [24] into [24, 61]
merged [45] and [30] into [30, 45]
merged [24, 61] and [30, 45] into [24, 30, 45, 61]
merged [9, 17, 38, 52] and [24, 30, 45, 61] into [9, 17, 24, 30, 38, 45, 52, 61]
[9, 17, 24, 30, 38, 45, 52, 61]
```

The merges are the same ones as in the 4.17.7 diagram, but look at the **order**. The diagram drew one level at a time, while the program finishes the entire left half, through `[9, 17, 38, 52]`, before it even starts splitting the right half. The reason is line 17: the call `mergeSort(left)` must return, fully sorted, before line 18 runs. Here is the start of the trace:

Step	Call running on	What happens
1	<code>[52, 17, 38, 9, 61, 24, 45, 30]</code>	splits; line 17 calls <code>mergeSort</code> on <code>[52, 17, 38, 9]</code> and waits
2	<code>[52, 17, 38, 9]</code>	splits; line 17 calls <code>mergeSort</code> on <code>[52, 17]</code> and waits
3	<code>[52, 17]</code>	splits; calls on <code>[52]</code> (base case) and on <code>[17]</code> (base case)
4	<code>[52, 17]</code>	line 19 merges: first output line
5	<code>[52, 17, 38, 9]</code>	line 18 calls on <code>[38, 9]</code> , which merges: second output line
6	<code>[52, 17, 38, 9]</code>	line 19 merges its two halves: third output line
7	whole array	line 17 has finally returned; line 18 starts on <code>[61, 24, 45, 30]</code>

Because `nums` is passed as a reference (Topic 3.6), line 19's `merge` changes the caller's array. That is how a sorted `left` reaches the call one level up: the caller's `left` array **is** the `nums` of the call below it.

The same method handles odd lengths. Running `mergeSort` on `{6, 2, 9, 4, 1}` from 4.17.7 prints:

```
merged [6] and [2] into [2, 6]
merged [4] and [1] into [1, 4]
merged [9] and [1, 4] into [1, 4, 9]
merged [2, 6] and [1, 4, 9] into [1, 2, 4, 6, 9]
[1, 2, 4, 6, 9]
```

The left part `[6, 2]` is finished first. Then the right part `[9, 4, 1]` splits into `[9]` and `[4, 1]`; `[9]` is a base case, `[4, 1]` is sorted to `[1, 4]`, and the two merge into `[1, 4, 9]`. The last merge combines `[2, 6]` with `[1, 4, 9]`.

Merge sort works on an `ArrayList` too, with `get`, `set`, and `size()` in place of brackets and `length`. The splitting and merging steps are the same.

WATCH OUT

A base case that lets one-element arrays through. A student writes `nums.length > 0` on line 5, reasoning that any non-empty array should be sorted:

MergeSortZeroBase.java

```
1 public class MergeSortZeroBase
2 {
3     public static void mergeSort(int[] nums)
4     {
5         if (nums.length > 0)
6         {
7             int[] left = new int[nums.length / 2];
8             int[] right = new int[nums.length - left.length];
9             for (int i = 0; i < left.length; i++)
10            {
11                left[i] = nums[i];
12            }
13            for (int i = 0; i < right.length; i++)
14            {
15                right[i] = nums[left.length + i];
16            }
17            mergeSort(left);
18            mergeSort(right);
19            merge(nums, left, right);
20            System.out.println("merged " + show(left) + " and " + show(right) +
21                " into " + show(nums));
22        }
23    }
24    public static void merge(int[] result, int[] left, int[] right)
25    {
26        int i = 0;
27        int j = 0;
28        int k = 0;
29        while (i < left.length && j < right.length)
30        {
31            if (left[i] <= right[j])
32            {
33                result[k] = left[i];
34                i++;
35            }
36            else
37            {
38                result[k] = right[j];
39                j++;
40            }
41            k++;
42        }
43        while (i < left.length)
44        {
```

```

45         result[k] = left[i];
46         i++;
47         k++;
48     }
49     while (j < right.length)
50     {
51         result[k] = right[j];
52         j++;
53         k++;
54     }
55 }
56
57 public static String show(int[] nums)
58 {
59     String text = "[";
60     for (int i = 0; i < nums.length; i++)
61     {
62         if (i > 0)
63         {
64             text = text + ", ";
65         }
66         text = text + nums[i];
67     }
68     return text + "]";
69 }
70
71 public static void main(String[] args)
72 {
73     int[] scores = {52, 17, 38, 9, 61, 24, 45, 30};
74     mergeSort(scores);
75     System.out.println(show(scores));
76 }
77 }

```

Exception in thread "main" java.lang.StackOverflowError

Follow an array of length 1. It passes the test, so `left` gets $1 / 2 = 0$ elements and `right` gets 1. The call on `right` receives another array of length 1, which does the same thing, forever. The split never makes the array smaller, so the recursion never stops and nothing is ever printed. The test must be `nums.length > 1`, so a one-element array is treated as the base case.

PAUSE AND TRY

Question 17 Tracing

`MergeSortFive.java` is `MergeSortDemo.java` with one change: `main` sorts `{8, 3, 5, 1, 9}`. Write the exact output.

MergeSortFive.java

```
1 public class MergeSortFive
2 {
3     public static void mergeSort(int[] nums)
4     {
5         if (nums.length > 1)
6         {
7             int[] left = new int[nums.length / 2];
8             int[] right = new int[nums.length - left.length];
9             for (int i = 0; i < left.length; i++)
10            {
11                left[i] = nums[i];
12            }
13            for (int i = 0; i < right.length; i++)
14            {
15                right[i] = nums[left.length + i];
16            }
17            mergeSort(left);
18            mergeSort(right);
19            merge(nums, left, right);
20            System.out.println("merged " + show(left) + " and " + show(right) +
    " into " + show(nums));
21        }
22    }
23
24    public static void merge(int[] result, int[] left, int[] right)
25    {
26        int i = 0;
27        int j = 0;
28        int k = 0;
29        while (i < left.length && j < right.length)
30        {
31            if (left[i] <= right[j])
32            {
33                result[k] = left[i];
34                i++;
35            }
36            else
37            {
38                result[k] = right[j];
39                j++;
40            }
41            k++;
42        }
43        while (i < left.length)
44        {
45            result[k] = left[i];
46            i++;
47            k++;
48        }
49        while (j < right.length)
50        {
51            result[k] = right[j];
```

```

52         j++;
53         k++;
54     }
55 }
56
57 public static String show(int[] nums)
58 {
59     String text = "[";
60     for (int i = 0; i < nums.length; i++)
61     {
62         if (i > 0)
63         {
64             text = text + ", ";
65         }
66         text = text + nums[i];
67     }
68     return text + "]";
69 }
70
71 public static void main(String[] args)
72 {
73     int[] scores = {8, 3, 5, 1, 9};
74     mergeSort(scores);
75     System.out.println(show(scores));
76 }
77 }

```

Question 18 Debugging

This version should sort {7, 3, 8, 1}, but its final line is [7, 7, 7, 7]. Find the line with the error, explain what goes wrong, and write the corrected line.

MergeSortCopyBug.java

```

1  public class MergeSortCopyBug
2  {
3      public static void mergeSort(int[] nums)
4      {
5          if (nums.length > 1)
6          {
7              int[] left = new int[nums.length / 2];
7              int[] right = new int[nums.length - left.length];
9              for (int i = 0; i < left.length; i++)
10             {
11                 left[i] = nums[i];
12             }
13             for (int i = 0; i < right.length; i++)
14             {
15                 right[i] = nums[i];
16             }
17             mergeSort(left);
18             mergeSort(right);
19             merge(nums, left, right);

```

```
20         System.out.println("merged " + show(left) + " and " + show(right) +  
    " into " + show(nums));  
21     }  
22 }  
23  
24 public static void merge(int[] result, int[] left, int[] right)  
25 {  
26     int i = 0;  
27     int j = 0;  
28     int k = 0;  
29     while (i < left.length && j < right.length)  
30     {  
31         if (left[i] <= right[j])  
32         {  
33             result[k] = left[i];  
34             i++;  
35         }  
36         else  
37         {  
38             result[k] = right[j];  
39             j++;  
40         }  
41         k++;  
42     }  
43     while (i < left.length)  
44     {  
45         result[k] = left[i];  
46         i++;  
47         k++;  
48     }  
49     while (j < right.length)  
50     {  
51         result[k] = right[j];  
52         j++;  
53         k++;  
54     }  
55 }  
56  
57 public static String show(int[] nums)  
58 {  
59     String text = "[";  
60     for (int i = 0; i < nums.length; i++)  
61     {  
62         if (i > 0)  
63         {  
64             text = text + ", ";  
65         }  
66         text = text + nums[i];  
67     }  
68     return text + "]";  
69 }  
70  
71 public static void main(String[] args)
```

```

72     {
73         int[] scores = {7, 3, 8, 1};
74         mergeSort(scores);
75         System.out.println(show(scores));
76     }
77 }

```

```

merged [7] and [7] into [7, 7]
merged [7] and [7] into [7, 7]
merged [7, 7] and [7, 7] into [7, 7, 7, 7]
[7, 7, 7, 7]

```

Putting it together

A school office keeps locker numbers in the order students signed up, which is not sorted. To use binary search, the office first sorts the numbers with merge sort, then searches them recursively. Before reading the trace, predict the sorted array and the indexes that each search checks.

WORKED EXAMPLE Sort the lockers, then search them

`mergeSort` and `merge` are the same as in `MergeSortDemo.java`, without the line that prints each merge. `binarySearch` is the recursive version from 4.17.5, printing the index and value it checks.

`LockerFinder.java`

```

1  public class LockerFinder
2  {
3      public static void mergeSort(int[] nums)
4      {
5          if (nums.length > 1)
6          {
7              int[] left = new int[nums.length / 2];
7              int[] right = new int[nums.length - left.length];
9              for (int i = 0; i < left.length; i++)
10             {
11                 left[i] = nums[i];
12             }
13             for (int i = 0; i < right.length; i++)
14             {
15                 right[i] = nums[left.length + i];
16             }
17             mergeSort(left);
18             mergeSort(right);
19             merge(nums, left, right);
20         }
21     }
22
23     public static void merge(int[] result, int[] left, int[] right)
24     {
25         int i = 0;

```

```
26     int j = 0;
27     int k = 0;
28     while (i < left.length && j < right.length)
29     {
30         if (left[i] <= right[j])
31         {
32             result[k] = left[i];
33             i++;
34         }
35         else
36         {
37             result[k] = right[j];
38             j++;
39         }
40         k++;
41     }
42     while (i < left.length)
43     {
44         result[k] = left[i];
45         i++;
46         k++;
47     }
48     while (j < right.length)
49     {
50         result[k] = right[j];
51         j++;
52         k++;
53     }
54 }
55
56 public static String show(int[] nums)
57 {
58     String text = "[";
59     for (int i = 0; i < nums.length; i++)
60     {
61         if (i > 0)
62         {
63             text = text + ", ";
64         }
65         text = text + nums[i];
66     }
67     return text + "]";
68 }
69
70 public static int binarySearch(int[] nums, int target, int low, int high)
71 {
72     if (low > high)
73     {
74         return -1;
75     }
76     int mid = (low + high) / 2;
77     System.out.println(" check index " + mid + ": " + nums[mid]);
78     if (nums[mid] == target)
79     {
```

```

80         return mid;
81     }
82     else if (nums[mid] < target)
83     {
84         return binarySearch(nums, target, mid + 1, high);
85     }
86     else
87     {
88         return binarySearch(nums, target, low, mid - 1);
89     }
90 }
91
92 public static void main(String[] args)
93 {
94     int[] lockers = {214, 105, 330, 187, 262, 149};
95     mergeSort(lockers);
96     System.out.println("Sorted: " + show(lockers));
97     System.out.println("Looking for 262");
98     System.out.println("Index: " + binarySearch(lockers, 262, 0,
lockers.length - 1));
99     System.out.println("Looking for 200");
100    System.out.println("Index: " + binarySearch(lockers, 200, 0,
lockers.length - 1));
101 }
102 }

```

Sorting. {214, 105, 330, 187, 262, 149} splits into [214, 105, 330] and [187, 262, 149].

```

                [214, 105, 330, 187, 262, 149]
split:         [214, 105, 330]         [187, 262, 149]
split:         [214]   [105, 330]       [187]   [262, 149]
split:         [105]   [330]           [262]   [149]
merge:         [214]   [105, 330]       [187]   [149, 262]
merge:         [105, 214, 330]         [149, 187, 262]
merge:         [105, 149, 187, 214, 262, 330]

```

Searching for 262. The sorted array has indexes 0 to 5.

Call	low	high	mid	Value	Decision
1	0	5	2	187	too small: right
2	3	5	4	262	found: return 4

Searching for 200.

Call	low	high	mid	Value	Decision
1	0	5	2	187	too small: right
2	3	5	4	262	too large: left
3	3	3	3	214	too large: left
4	3	2			low > high: return -1

```
Sorted: [105, 149, 187, 214, 262, 330]
Looking for 262
  check index 2: 187
  check index 4: 262
Index: 4
Looking for 200
  check index 2: 187
  check index 4: 262
  check index 3: 214
Index: -1
```

Three things to notice:

- The sort is what makes the search valid. On the original order, the first check would land on 330 at index 2, and a search for 262 would go left and never find it.
- Both algorithms are recursive, but in different shapes: binary search makes **one** recursive call per call and passes the answer straight back, while merge sort makes **two** recursive calls and then does more work (the merge) after they return.
- The search for 200 ends at the spot where 200 would belong, between 187 and 214, just as the search for 10 did in 4.17.3.

Vocabulary

Term	Definition	Example
recursive traversal	Visiting every character or element of a String, array, or ArrayList by handling one position per recursive call	<code>sumFrom(nums, index + 1)</code>
binary search	A search that checks the middle of the remaining range of sorted data and eliminates the half that cannot hold the target	Finding 26 in 3 checks among 9 ages
merge sort	A recursive sort that splits an array into one-element subarrays, then merges sorted subarrays back together in order	<code>{6, 2, 9, 4, 1}</code> becomes <code>{1, 2, 4, 6, 9}</code>
subarray	A piece of an array made of some of its elements in their original order	<code>[52, 17]</code> from <code>[52, 17, 38, 9]</code>
merge	To combine two sorted lists into one sorted list by repeatedly taking the smaller next element	<code>[17, 52]</code> and <code>[9, 38]</code> give <code>[9, 17, 38, 52]</code>

Chapter summary

- Recursion can traverse a String, an array, or an ArrayList. Each call handles one position and passes a shorter String (`substring(1)`) or the next index (`index + 1`); the base case is reached when nothing is left. A recursive call that makes no progress causes a `StackOverflowError`.

- Binary search requires sorted data. It checks the middle of the range from `low` to `high` and eliminates half of the remaining elements with each check, until it finds the target or `low` passes `high`. On unsorted data it can report a value missing that is really there.
- Binary search can be written iteratively, with a `while (low <= high)` loop, or recursively, with `low` and `high` as parameters and two base cases. Both check the same elements in the same order.
- Binary search usually needs far fewer checks than linear search, because doubling the data adds only one check to its worst case. It is not always faster; linear search wins when the target is near the front.
- Merge sort is a recursive sort for arrays and `ArrayList`s. It splits until every subarray has one element, then merges sorted subarrays in pairs, always taking the smaller next value and copying any leftovers. The program finishes the whole left half before it starts on the right half.

Review questions

Question 19 Multiple choice

This recursive binary search works on an `ArrayList<String>` sorted alphabetically. It uses `compareTo` (Topic 1.15) to decide which way to go. What does the program print?

NameSearch.java

```
1 import java.util.ArrayList;
2
3 public class NameSearch
4 {
5     public static int search(ArrayList<String> names, String target, int low, int
high)
6     {
7         if (low > high)
8         {
9             return -1;
10        }
11        int mid = (low + high) / 2;
12        System.out.println("compare with " + names.get(mid));
13        int result = target.compareTo(names.get(mid));
14        if (result == 0)
15        {
16            return mid;
17        }
18        else if (result > 0)
19        {
20            return search(names, target, mid + 1, high);
21        }
22        else
23        {
24            return search(names, target, low, mid - 1);
25        }
26    }
27
28    public static void main(String[] args)
29    {
30        ArrayList<String> roster = new ArrayList<String>();
```

```

31     roster.add("Ava");
32     roster.add("Ben");
33     roster.add("Cruz");
34     roster.add("Dev");
35     roster.add("Eli");
36     roster.add("Fay");
37     roster.add("Gus");
38     System.out.println(search(roster, "Bo", 0, roster.size() - 1));
39 }
40 }

```

- A. compare with Dev, compare with Ben, compare with Cruz, then -1
- B. compare with Dev, compare with Ben, then -1
- C. compare with Ava, compare with Ben, compare with Cruz, then -1
- D. compare with Dev, compare with Cruz, compare with Ben, then -1

Question 20 Tracing

Each call checks the first and last letters, then passes the middle of the word to the next call. Write the exact output.

PalindromeCheck.java

```

1  public class PalindromeCheck
2  {
3      public static boolean isPalindrome(String word)
4      {
5          System.out.println("checking " + word);
6          if (word.length() <= 1)
7          {
8              return true;
9          }
10         String first = word.substring(0, 1);
11         String last = word.substring(word.length() - 1);
12         if (!first.equals(last))
13         {
14             return false;
15         }
16         return isPalindrome(word.substring(1, word.length() - 1));
17     }
18
19     public static void main(String[] args)
20     {
21         System.out.println(isPalindrome("racecar"));
22         System.out.println(isPalindrome("rocker"));
23     }
24 }

```

Question 21 Multiple choice

What does this program print?

MidSequence.java

```

1  public class MidSequence
2  {

```

```

3     public static int binarySearch(int[] nums, int target)
4     {
5         int low = 0;
6         int high = nums.length - 1;
7         while (low <= high)
8         {
9             int mid = (low + high) / 2;
10            System.out.println("compare with " + nums[mid]);
11            if (nums[mid] == target)
12            {
13                return mid;
14            }
15            else if (nums[mid] < target)
16            {
17                low = mid + 1;
18            }
19            else
20            {
21                high = mid - 1;
22            }
23        }
24        return -1;
25    }
26
27    public static void main(String[] args)
28    {
29        int[] evens = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24};
30        System.out.println(binarySearch(evens, 15));
31    }
32 }

```

- A. compare with 12, compare with 18, compare with 14, compare with 16, then -1
- B. compare with 12, compare with 18, compare with 14, then -1
- C. compare with 14, compare with 18, compare with 16, then -1
- D. compare with 12, compare with 18, compare with 14, compare with 16, then 7

Question 22 Short answer

A student has an **unsorted** array of 1000 numbers and needs to find out, just once, whether 742 is in it. They plan to run binary search on the array as it is. Explain the problem with that plan. Then explain why sorting first and then using binary search is not a good trade for a single search.

Question 23 Tracing

This method compares values on the way **back up** from the recursion. Write the exact output, and give the value `maxFrom` returns for each index during `maxFrom(temps, 0)`.

MaxFrom.java

```
1 public class MaxFrom
2 {
3     public static int maxFrom(int[] nums, int index)
4     {
5         if (index == nums.length - 1)
6         {
7             return nums[index];
8         }
9         int restMax = maxFrom(nums, index + 1);
10        if (nums[index] > restMax)
11        {
12            return nums[index];
13        }
14        return restMax;
15    }
16
17    public static void main(String[] args)
18    {
19        int[] temps = {3, 9, 2, 6};
20        System.out.println(maxFrom(temps, 0));
21        System.out.println(maxFrom(temps, 2));
22    }
23 }
```

Question 24 Multiple choice

mergeSort from MergeSortDemo.java is called on {29, 4, 17, 50, 11, 36, 2}. Which two arrays are combined in the **last** merge?

- A. [4, 17, 29] and [2, 11, 36, 50]
- B. [29, 4, 17] and [50, 11, 36, 2]
- C. [2, 4, 11, 17] and [29, 36, 50]
- D. [4, 17, 29, 50] and [2, 11, 36]

Question 25 Multiple choice

Which statement about the searching and sorting algorithms in Topics 4.14 through 4.17 is true?

- A. Selection sort and insertion sort are recursive, and merge sort is iterative.
- B. Binary search can be written with a loop or with recursion, and either way it requires sorted data.
- C. Linear search, like binary search, works only on sorted data.
- D. Merge sort can sort an array but not an ArrayList.

Reflection

- ☐ I can trace a recursive method that works through a String one character at a time. Reread: 4.17.1 Recursion on Strings
- ☐ I can trace a recursive method that works through an array or ArrayList using an index parameter, and spot a base case that stops too late. Reread: 4.17.2 Recursion on Arrays and ArrayLists
- ☐ I can explain why binary search needs sorted data and list the elements it checks, step by step. Reread: 4.17.3 The Binary Search Idea

- ☐ I can trace low, high, and mid through each pass of an iterative binary search. Reread: 4.17.4
Binary Search with a Loop
- ☐ I can trace the calls of a recursive binary search and match its base cases to the parts of the loop version. Reread: 4.17.5 Binary Search with Recursion
- ☐ I can explain why binary search is typically, but not always, faster than linear search. Reread: 4.17.6 Comparing Binary Search and Linear Search
- ☐ I can draw how merge sort splits an array into one-element pieces and merges them back. Reread: 4.17.7 Merge Sort: Split, Then Merge
- ☐ I can trace the merge of two sorted arrays, including the leftover values. Reread: 4.17.8 Merging Two Sorted Halves
- ☐ I can give the order in which the recursive mergeSort method performs its merges. Reread: 4.17.9 The Recursive mergeSort Method

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
pae
rbt
```

`substring(2)` drops the first **two** characters, so each call keeps one letter and skips the next.

text	Returns
""	"" (base case, length 0)
"et"	"e" + "" = "e"
"anet"	"a" + "e" = "ae"
"planet"	"p" + "ae" = "pae"

For "robot": "robot" keeps `r` and passes "bot", which keeps `b` and passes "t". "t" has length 1, so it is a base case and returns "t". The result is "rbt". The base case covers length 1 as well as 0, which matters here: `"t".substring(2)` would crash, so the method must stop before trying it.

Reread 4.17.1 *Recursion on Strings* if your trace differed.

Question 2

C. One call for each length from 4 down to 0: "moon", "oon", "on", "n", and "". The call with "" is the base case.

- **A** is wrong: 2 is the value returned (the number of `os`), not the number of calls.
- **B** is wrong: it forgets the base-case call with the empty string. That call really happens; it is the one that returns 0.
- **D** is wrong: there is no call after ""; the base case returns without calling again.

Reread 4.17.1 *Recursion on Strings* if you chose differently.

Question 3

```
5
30
12
```

Call	index	What it does
1	0	calls with 1 and waits; prints 12 after it returns
2	1	calls with 2 and waits; prints 30 after it returns
3	2	calls with 3 and waits; prints 5 after it returns
4	3	<code>3 < 3</code> is false, so it does nothing and returns

Call 4 returns first, so call 3 prints first (5), then call 2 (30), then call 1 (12). The method visits the elements front to back but prints on the way back up, so the output is in reverse order. If the `println` came before the recursive call, the output would be 12, 30, 5.

Reread 4.17.2 *Recursion on Arrays and ArrayLists* if your output differed.

Question 4

A. The traversal starts at the index you pass and runs to the end, so starting at 0 includes every element: $6 + 2 + 9 + 5 = 22$.

- **B** is wrong: starting at 1 skips the 6 and returns 16.
- **C** is wrong: `prices.length` is 4, which is the base case, so it returns 0 right away.
- **D** is wrong: index 3 is the last element, so it returns only 5.

Reread 4.17.2 *Recursion on Arrays and ArrayLists* if you chose differently.

Question 5

(a) 36: compares with 23, then 36; found at index 5.

low	high	mid	Value	Decision
0	6	3	23	too small: low = 4
4	6	5	36	found at index 5

(b) 5: compares with 23, 11, then 4; not found.

low	high	mid	Value	Decision
0	6	3	23	too large: high = 2
0	2	1	11	too large: high = 0
0	0	0	4	too small: low = 1
1	0			low past high: not found

Reread 4.17.3 *The Binary Search Idea* if your list differed.

Question 6

B. It is the only array in smallest-to-largest order. Repeated values are fine, because each element is still no smaller than the one before it.

- **A** is wrong: 12 comes before the smaller 9, so the array is not sorted.
- **C** is wrong: 40 comes first but is bigger than three of the values after it. The middle element, 25, is in a reasonable spot, which might fool you, but searching for 40 would go right from 25 and miss it.
- **D** is wrong: 8, 6, 4 go down and then 9, 12 go up, so it is not sorted.

Reread 4.17.3 *The Binary Search Idea* if you chose differently.

Question 7

```
0 9 4
5 9 7
8 9 8
8
0 9 4
0 3 1
0 0 0
-1
```

Searching for 52: `times[4]` is 23, too small, so `low = 5`. `mid = (5 + 9) / 2 = 7`, and `times[7]` is 44, too small, so `low = 8`. `mid = (8 + 9) / 2 = 8`, and `times[8]` is 52: return 8.

Searching for 4: `times[4]` is 23, too large, so `high = 3`. `mid = 1`, and 9 is too large, so `high = 0`. `mid = 0`, and 5 is too large, so `high = -1`. Now `0 <= -1` is false, the loop ends, and the method returns -1. The loop printed only three lines, because the check that ends the loop happens before line 10.

Reread 4.17.4 *Binary Search with a Loop* if your output differed.

Question 8

Line 14 has the comparison backward. It says that when the middle value is **larger** than the target, the search should move right (`low = mid + 1`). But a larger middle value means the target is to the **left**.

Searching for 160: `heights[3]` is 140. Since 140 is not greater than 160, the `else` runs and sets `high` to 2, throwing away the right half, where 160 actually is. The search then wanders through indexes 0 to 2 and returns -1. Searching for 140 only worked by luck: 140 was the first middle element checked, so the broken line never ran.

Corrected line 14:

```
1         else if (nums[mid] < target)
```

With that fix the program prints 3 and 5.

Reread 4.17.4 *Binary Search with a Loop* if you missed it.

Question 9

A. Call 1 (`low` 0, `high` 8) checks `ages[4]` = 21, too small, so call 2 gets `mid + 1 = 5` and `high` 8. Call 2 checks `ages[6]` = 30, too small, so call 3 gets 7 and 8. Call 3 checks `ages[7]` = 34 and returns 7. Three calls.

- **B** is wrong: the new `low` is `mid + 1`, not `mid`, because index 4 was already checked.
- **C** is wrong: the search ends as soon as `nums[mid] == target`; no extra call reaches `low > high`.
- **D** is wrong: 21 is less than 34, so the search goes right, not left.

Reread 4.17.5 *Binary Search with Recursion* if you chose differently.

Question 10

Model answer: The base case `if (low > high) return -1`; plays the role of the loop condition: when the range is empty, the recursion stops instead of the loop. The `return` in front of each recursive call passes the answer from the deeper call back up to the caller. The call that finds the target returns its index to the call before it, and that call must return the same value to its own caller, all the way back to `main`.

A complete answer says:

- The `low > high` base case replaces the loop condition (both mean "nothing left to search").
- Each call must return the result of its recursive call so the index found deep in the recursion reaches the original caller.

Reread 4.17.5 *Binary Search with Recursion* if your answer was missing one of these.

Question 11

A. Halving from 100: $100 \rightarrow 50 \rightarrow 25 \rightarrow 12 \rightarrow 6 \rightarrow 3 \rightarrow 1$ is six halvings, plus a check of the last remaining element, for at most 7 checks.

- **B** is wrong: 10 is the worst case for 1000 elements, not 100.
- **C** is wrong: it halves once and then imagines checking everything that is left, which is how linear search would behave on half the array.
- **D** is wrong: 100 is linear search's worst case.

Reread 4.17.6 *Comparing Binary Search and Linear Search* if you chose differently.

Question 12

Model answer: Binary search is typically faster, not always. It starts at the middle, so finding the very first element took it 9 checks, while linear search found it in 1. When nearly every target is at the front, linear search checks only 1 to 3 elements and wins. Binary search's advantage shows up when targets can be anywhere, or are missing, where it needs at most 10 checks instead of up to 1000.

A complete answer says:

- Binary search is typically, not always, more efficient.
- Linear search starts at the front, so it is very fast when the target is near the front (1 check versus 9 for id 0).

Reread 4.17.6 *Comparing Binary Search and Linear Search* if your answer was missing one of these.

Question 13

```

                [40, 15, 33, 8, 27, 12]
split:         [40, 15, 33]         [8, 27, 12]
split:         [40]   [15, 33]      [8]   [27, 12]
split:         [15]  [33]           [27]  [12]
merge:         [40]   [15, 33]      [8]   [12, 27]
merge:         [15, 33, 40]         [8, 12, 27]
merge:         [8, 12, 15, 27, 33, 40]
```

$6 / 2 = 3$, so each half has 3 elements. Each half of 3 splits into a left piece of $3 / 2 = 1$ and a right piece of 2. [15] and [33] merge to [15, 33] (already in order), and [27] and [12] merge to [12, 27]. Then [40] joins [15, 33], [8] joins [12, 27], and the final merge combines the two sorted halves.

Reread 4.17.7 *Merge Sort: Split, Then Merge* if your diagram differed.

Question 14

B. One element cannot be out of order, so it needs no sorting. That is the point where splitting stops and merging begins.

- **A** is wrong: splitting cuts at a position and keeps the original order; only merging compares values.
- **C** is wrong: merge sort sorts each half by splitting and merging it too. It never uses selection sort.
- **D** is wrong: odd lengths just give uneven pieces, as the 5-value example shows.

Reread 4.17.7 *Merge Sort: Split, Then Merge* if you chose differently.

Question 15

k	Compare	Copied	i, j after
0	4 vs 6	4	1, 0
1	10 vs 6	6	1, 1
2	10 vs 7	7	1, 2
3	10 vs 20	10	2, 2
4	15 vs 20	15	3, 2

Now `i` is 3, so `left` is used up and the first loop stops after five comparisons. The leftover loop for `right` copies 20 into `result[5]`.

```
[4, 6, 7, 10, 15, 20]
```

Reread 4.17.8 *Merging Two Sorted Halves* if your trace differed.

Question 16

C. `i` equals `left.length`, so `left` is used up and its leftover loop does not run. `right` still has indexes 1, 2, and 3 unused, so its leftover loop copies those 3 values.

- **A** is wrong: only 3 of the 6 values (2 from `left`, 1 from `right`) have been placed so far.
- **B** is wrong: `i < left.length` is `2 < 2`, which is `false`, so that loop does not run.
- **D** is wrong: `right[0]` was already copied during the first loop (that is why `j` is 1).

Reread 4.17.8 *Merging Two Sorted Halves* if you chose differently.

Question 17

```
merged [8] and [3] into [3, 8]
merged [1] and [9] into [1, 9]
merged [5] and [1, 9] into [1, 5, 9]
merged [3, 8] and [1, 5, 9] into [1, 3, 5, 8, 9]
[1, 3, 5, 8, 9]
```

$5 / 2 = 2$, so `left` is `[8, 3]` and `right` is `[5, 1, 9]`. The left part is sorted completely first: `[8]` and `[3]` merge into `[3, 8]`. Then `[5, 1, 9]` splits into `[5]` and `[1, 9]`; `[1, 9]` splits into `[1]` and `[9]`, which merge into `[1, 9]`; then `[5]` and `[1, 9]` merge into `[1, 5, 9]`. Last, the two halves merge.

Reread 4.17.9 *The Recursive mergeSort Method* if your output differed.

Question 18

Line 15 copies `nums[i]` into `right[i]`, so `right` gets the **first** elements of `nums`, the same ones `left` got, instead of the elements after them. For `{7, 3, 8, 1}`, both `left` and `right` become `{7, 3}`, and the 8 and 1 are lost. Inside those calls the same mistake happens again: `{7, 3}` splits into `[7]` and `[7]`. Every merge then combines copies of 7.

Corrected line 15:

```
1          right[i] = nums[left.length + i];
```

The right part starts where the left part ends, at index `left.length`. With that fix the program merges `[7]` and `[3]`, then `[8]` and `[1]`, then `[3, 7]` and `[1, 8]`, and prints `[1, 3, 7, 8]`.

Reread 4.17.9 *The Recursive mergeSort Method* if you missed it.

Question 19

A. The list has indexes 0 to 6.

Call	low	high	mid	Name	"Bo".compareTo(name)	Next
1	0	6	3	Dev	negative (B before D)	left: 0 to 2
2	0	2	1	Ben	positive (o after e)	right: 2 to 2
3	2	2	2	Cruz	negative (B before C)	left: 2 to 1
4	2	1				return -1

The fourth call prints nothing, because the base case on line 7 returns before line 12 runs.

- **B** is wrong: after `Ben`, the range 2 to 2 still holds one name, so the search must check `Cruz` before giving up.
- **C** is wrong: it checks names from the front, which is linear search.
- **D** is wrong: after `Dev`, the middle of indexes 0 to 2 is index 1, `Ben`, not `Cruz`.

Reread 4.17.4 *Binary Search with a Loop* (comparing Strings) and 4.17.5 *Binary Search with Recursion* if you chose differently.

Question 20

```
checking racecar
checking aceca
checking cec
checking e
true
checking rocker
checking ocke
false
```

`word.substring(1, word.length() - 1)` drops the first **and** last letters. For `"racecar"`: `r` matches `r`, so check `"aceca"`; `a` matches `a`, so check `"cec"`; `c` matches `c`, so check `"e"`, which has length 1 and is a base case returning `true`. Each waiting call returns that `true`.

For `"rocker"`: `r` matches `r`, so check `"ocke"`; `o` does not match `e`, so that call returns `false` right away without another recursive call. That `false` is passed back up.

Reread 4.17.1 *Recursion on Strings* if your output differed.

Question 21

A. The array has 12 elements, indexes 0 to 11.

low	high	mid	Value	Decision
0	11	5	12	too small: low = 6
6	11	8	18	too large: high = 7
6	7	6	14	too small: low = 7
7	7	7	16	too large: high = 6

Now `7 <= 6` is `false`, so the method returns `-1`. 15 is odd, so it cannot be in an array of even numbers.

- **B** is wrong: it stops when `low` and `high` are equal, but that range still holds one element (16), which must be checked.
- **C** is wrong: $(0 + 11) / 2$ is 5 with `int` division, so the first value checked is `evens[5] = 12`, not 14.
- **D** is wrong: 7 is the last index checked, but the value there is 16, not 15. The method returns an index only when the value matches.

Reread 4.17.4 *Binary Search with a Loop* if you chose differently.

Question 22

Model answer: Binary search's precondition is sorted data. On an unsorted array it throws away halves based on a false assumption, so it can report that 742 is missing when it is actually there, with no error to warn the student. Sorting first would make binary search correct, but any sort has to look at every one of the 1000 elements at least once, which is already as much work as linear search does in its worst case. For one search, linear search on the unsorted array is simpler and does no more work. Sorting pays off only when the data will be searched many times.

A complete answer says:

- Binary search requires sorted data; on unsorted data it can give a wrong answer.
- For a single search, the cost of sorting is at least as much as a linear search, so linear search is the better choice.

Reread 4.17.3 *The Binary Search Idea* and 4.17.6 *Comparing Binary Search and Linear Search* if your answer was missing one of these.

Question 23

9
6

For `maxFrom(temps, 0)` with `temps = {3, 9, 2, 6}`:

index	nums[index]	restMax	Returns
3	6	(base case)	6
2	2	6	6
1	9	6	9
0	3	9	9

The base case here is the **last** element (`index == nums.length - 1`), not one past it, because the largest value of a one-element range is that element. `maxFrom(temps, 2)` looks only at indexes 2 and 3, so it returns the larger of 2 and 6, which is 6.

Reread 4.17.2 *Recursion on Arrays and ArrayLists* if your trace differed.

Question 24

A. $7 / 2 = 3$, so `left` holds the first three values, `[29, 4, 17]`, and `right` holds the other four, `[50, 11, 36, 2]`. Each part is fully sorted by its recursive call before the last merge, so the merge combines `[4, 17, 29]` and `[2, 11, 36, 50]` into `[2, 4, 11, 17, 29, 36, 50]`.

- **B** is wrong: these are the parts right after the split, before the recursive calls sort them. `merge` needs sorted inputs.
- **C** is wrong: it splits the sorted result instead of the original array. The split happens on the original order.
- **D** is wrong: it gives the left part four elements, but `left` gets `length / 2 = 3` elements and the extra one goes to `right`.

Reread *4.17.7 Merge Sort: Split, Then Merge* and *4.17.9 The Recursive mergeSort Method* if you chose differently.

Question 25

B. The iterative and recursive versions check the same elements in the same order, and both depend on sorted data to know which half to throw away.

- **A** is wrong: it is the other way around. Selection sort and insertion sort use loops; merge sort is recursive.
- **C** is wrong: linear search checks every element in turn, so it works on data in any order.
- **D** is wrong: merge sort can sort an `ArrayList` too, using `get`, `set`, and `size()`.

Reread *4.17.5 Binary Search with Recursion* and *4.17.9 The Recursive mergeSort Method* if you chose differently.