AP CSA

zhang si 张思

zhangsi@rdfz.cn

ICC 609

7- recursion & search & sort

- recursion
- search
    - liner search
    - bineary search
- sort
    - selection sort
    - insertion sort
    - merge sort

What is Recursion?

- A recursive method is a method that calls **itself**.
- Every recursive method has **two** distinctive parts
  - ✓ A base case or termination condition that causes the method to end
  - ✓ A nonbase case whose actions move the algorithm toward the base case and termination

Recursion in Math

- The value referred to as N ! (pronounced N factorial) is defined for any positive integer N as the product of all integers between 1 and N inclusive. Therefore, 3! is defined as:

$$3! = 3*2*1 = 6$$

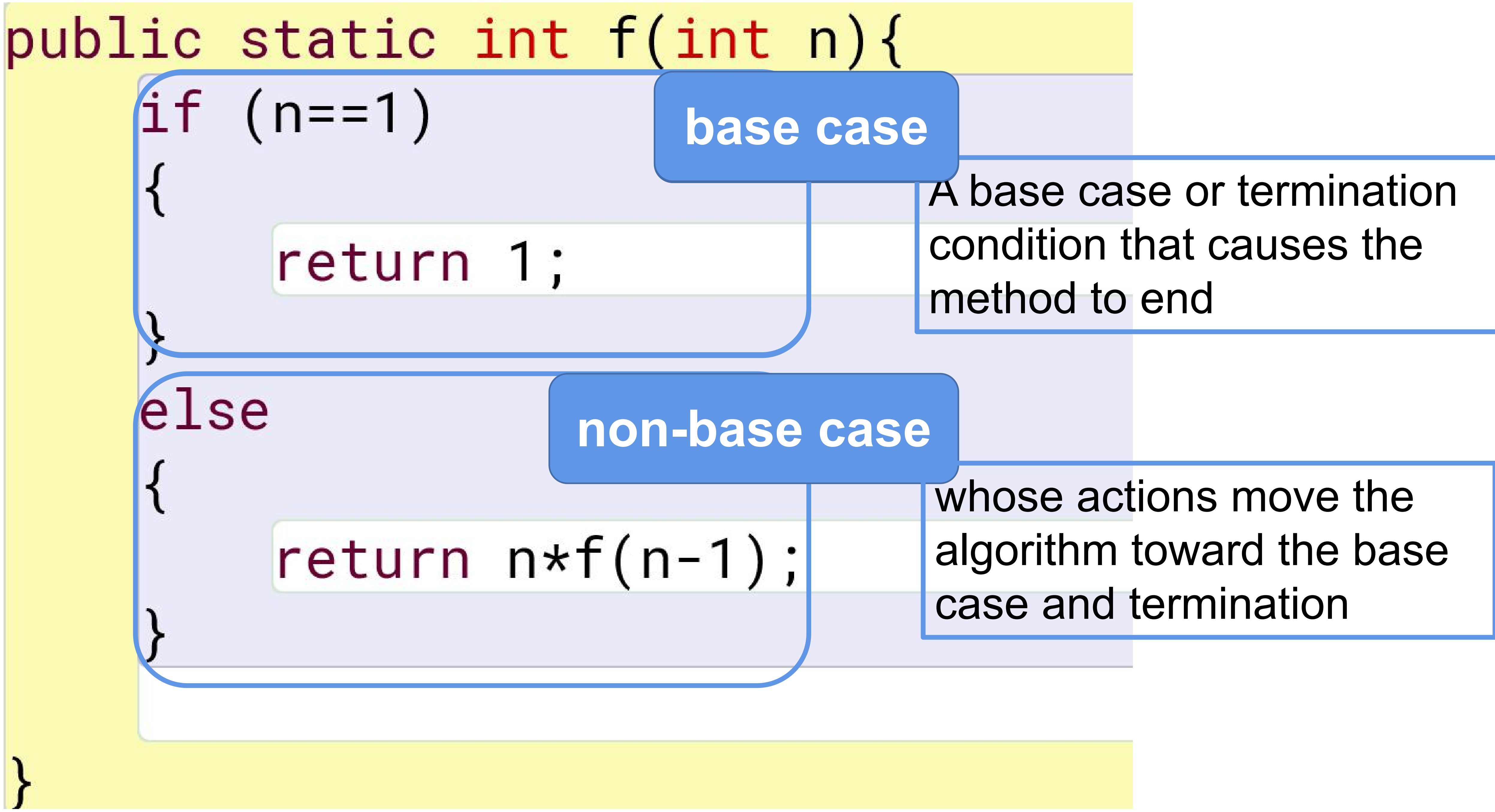
- and 5! is defined as:

$$5! = 5*4*3*2*1 = 120.$$

- Mathematical formulas are often expressed recursively. The definition of N !
- can be expressed recursively as:

$$1! = 1$$

$$N! = N * (N-1)! \text{ for } N > 1$$



```java
public static int f(int n){
        if (n==1)
        {
            return 1;
        }
        else
        {
            return n*f(n-1);
        }
}
```

base case

A base case or termination condition that causes the method to end

non-base case

whose actions move the algorithm toward the base case and termination

Fibonacci sequence

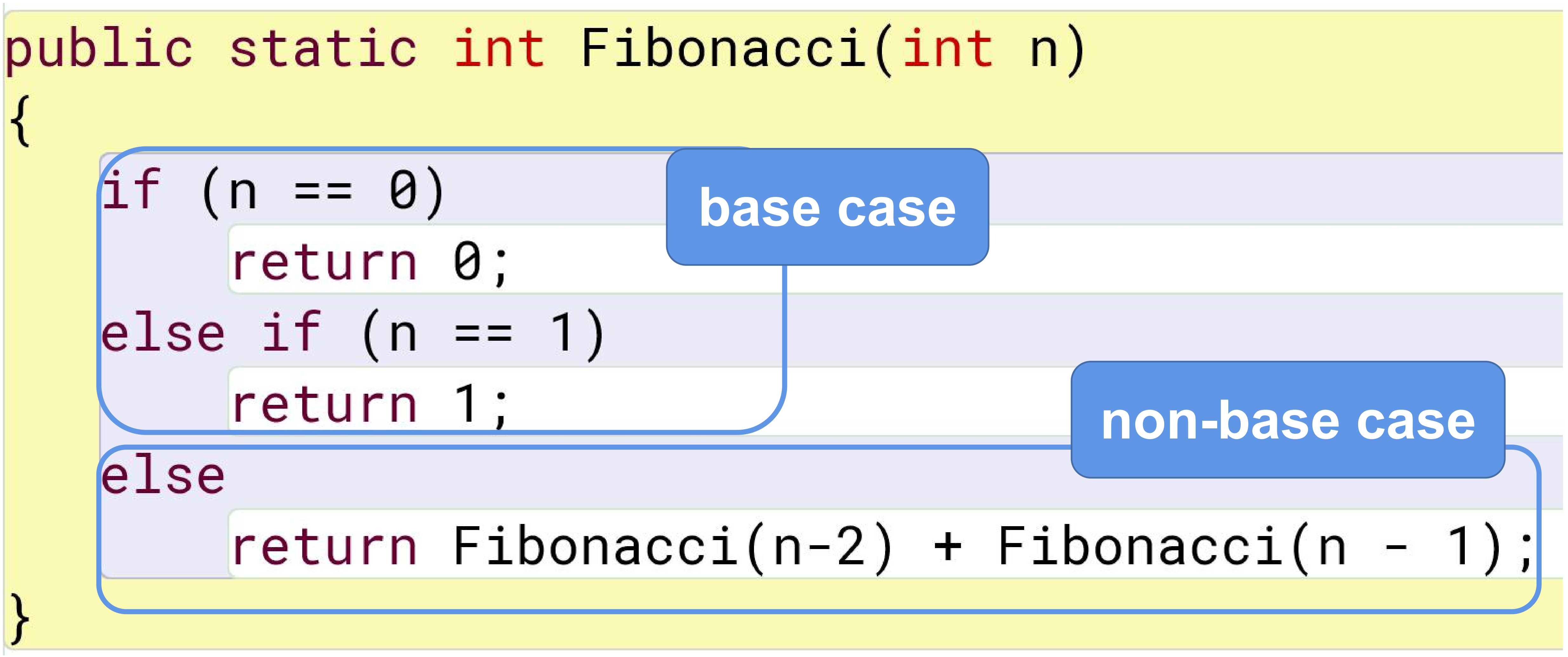
- Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

$$F_0=0,F_1=1$$

$$F_n=F_{n-1}+F_{n-2}$$

- for n > 1.
- The sequence starts:
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Fibonacci sequence



```
public static int Fibonacci(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return Fibonacci(n-2) + Fibonacci(n - 1);
}
```

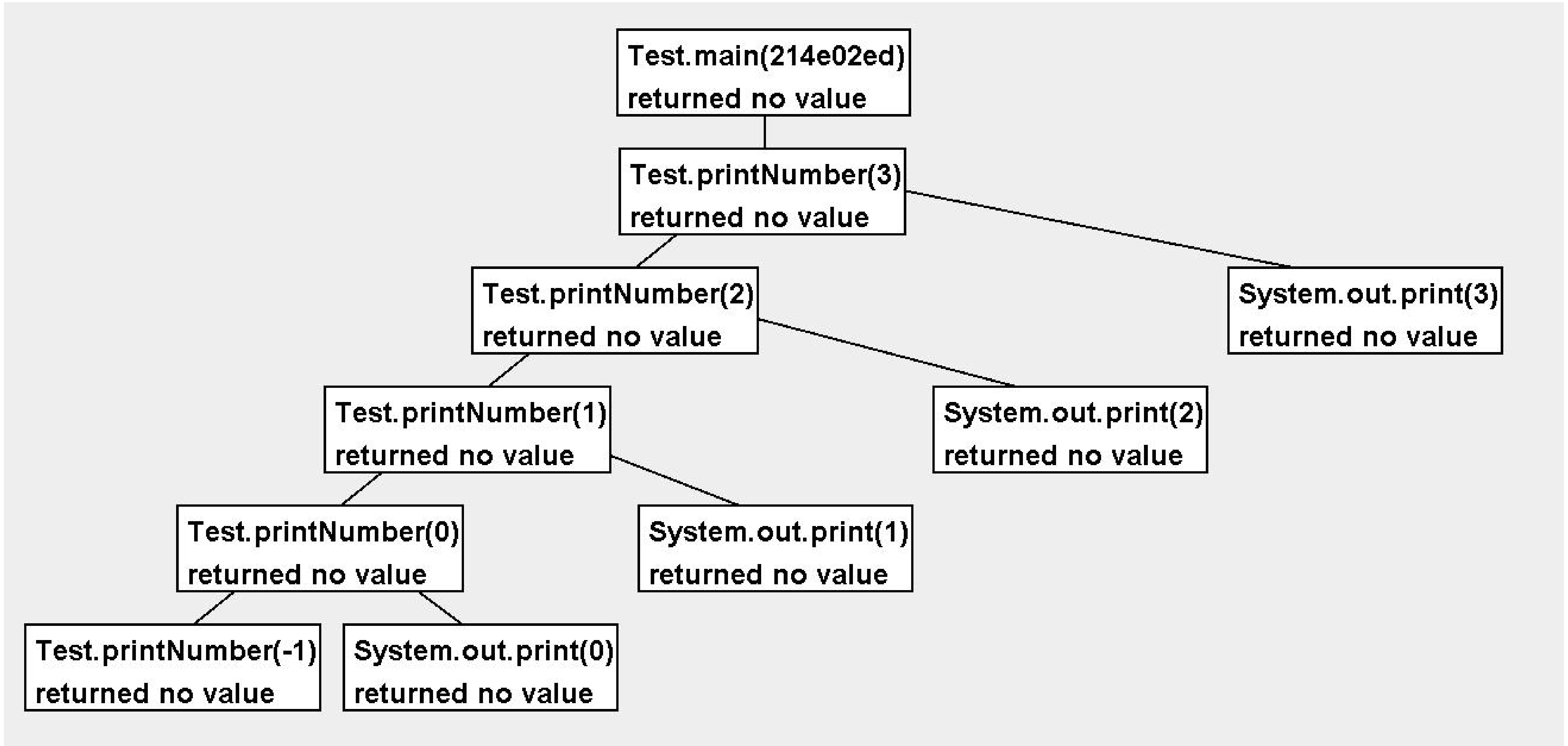
Tracing recursive methods

Given the method defined below what does the following print?

```
printNumber(3);
```

```
public static void printNumber (int n)
{
     if (n >= 0)
    {
             printNumber(n-1);
         System.out.print(n);
    }
}
```

Tracing recursive methods

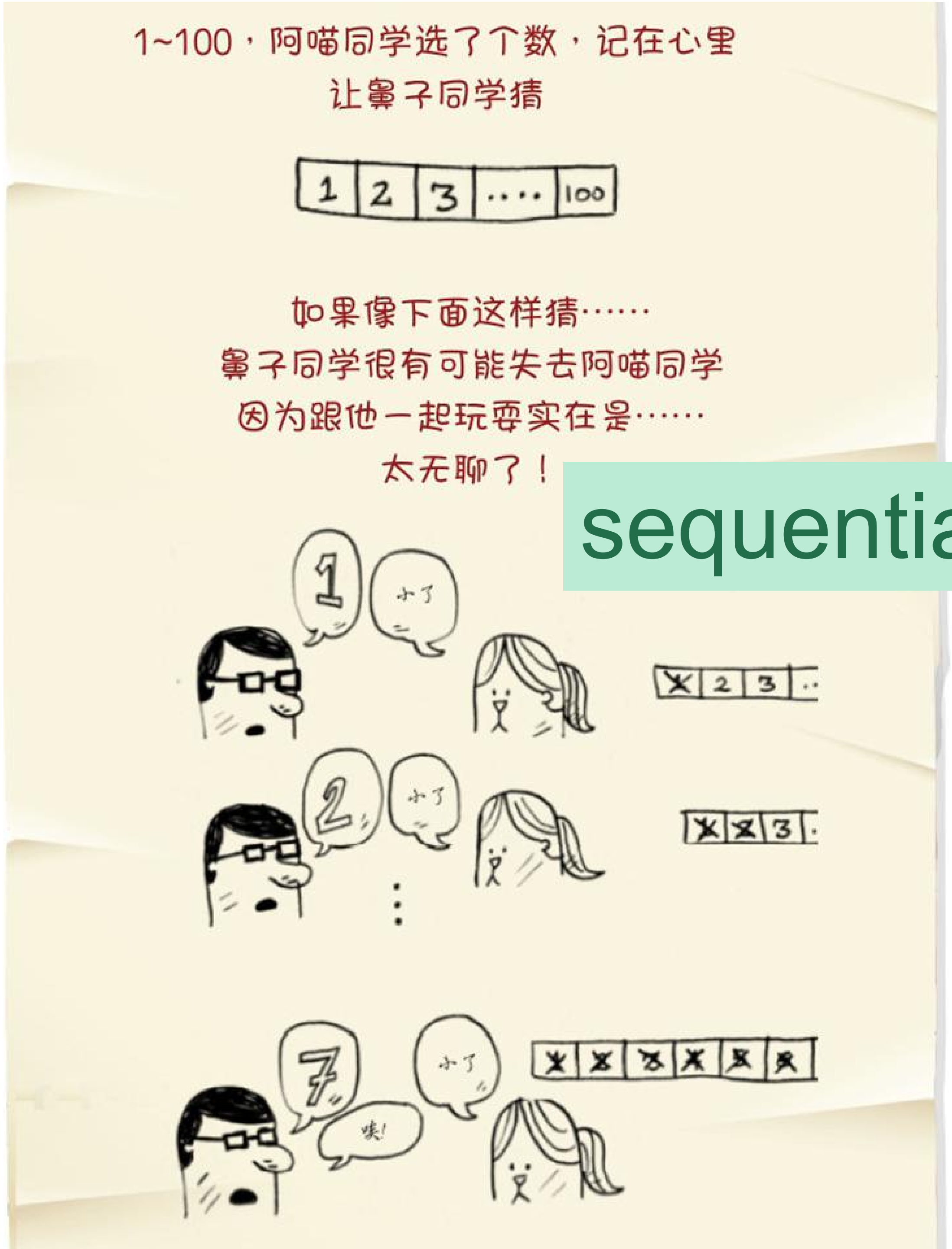


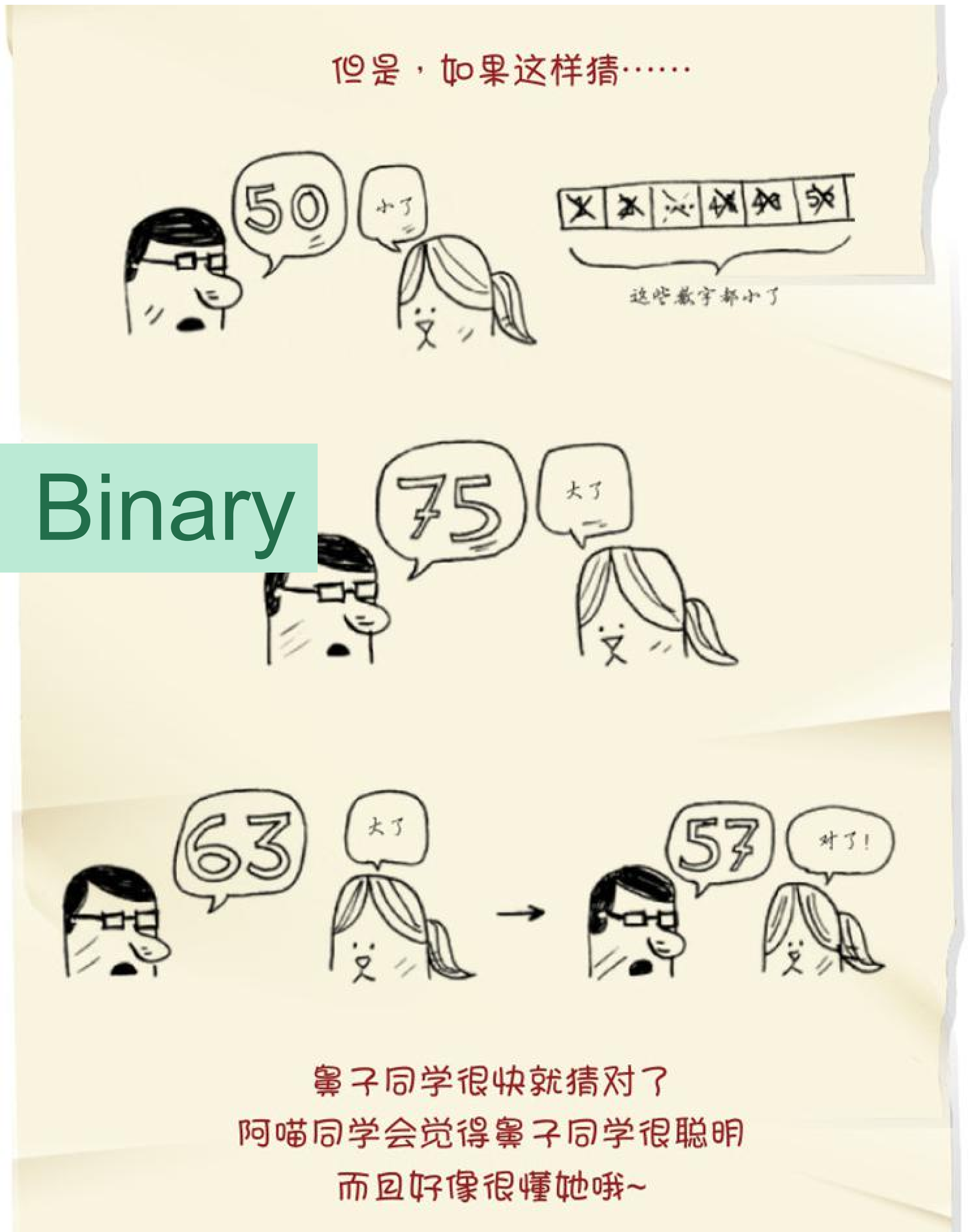
Recursion

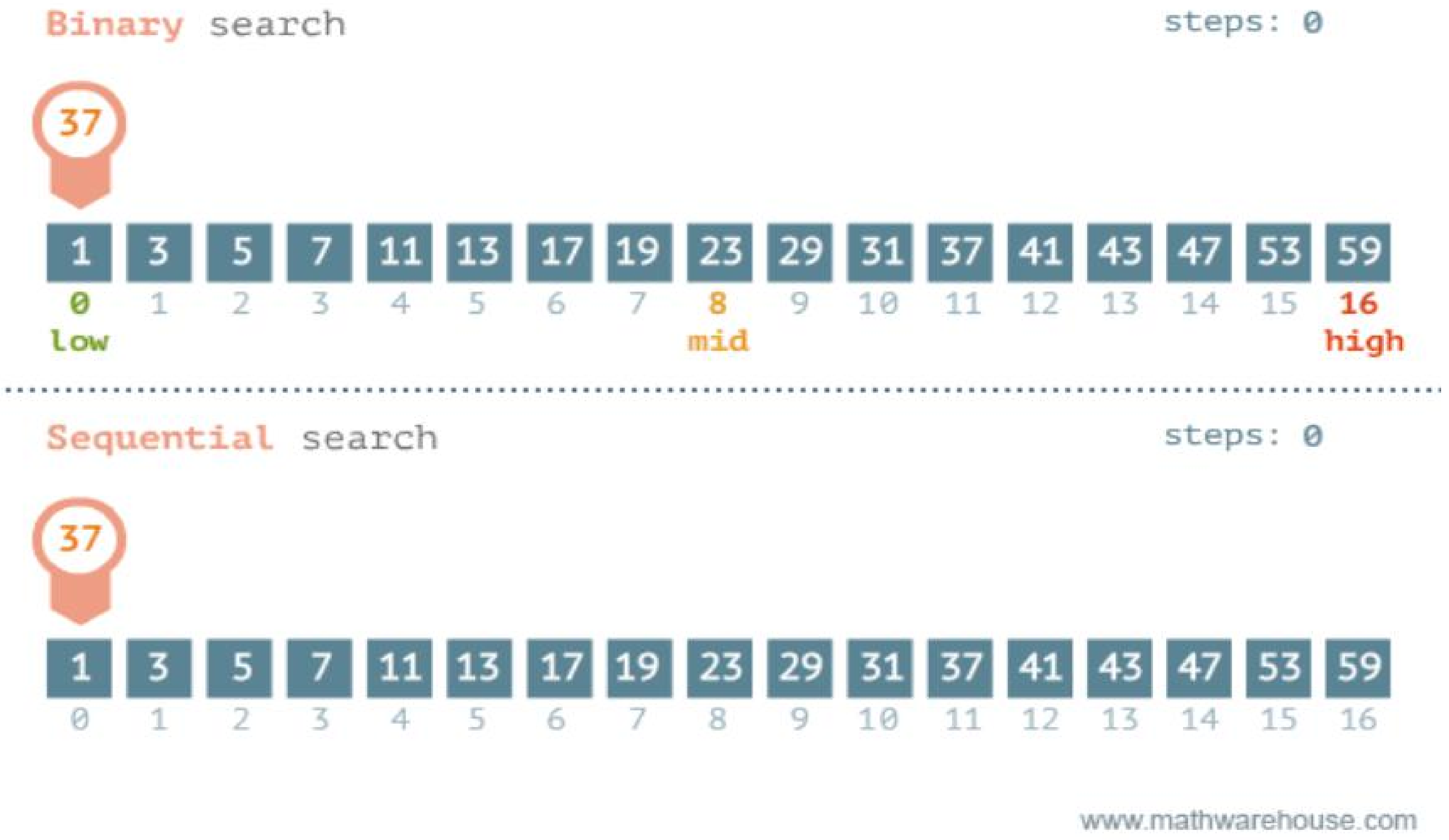
- ESSENTIAL KNOWLEDGE
- A recursive method is a method that calls itself.
- Recursive methods contain at least one base case, which halts the recursion, and at least one recursive call.
- Each recursive call has its own set of local variables, including the formal parameters.
- Parameter values capture the progress of a recursive process, much like loop control variable values capture the progress of a loop
- Any recursive solution can be replicated through the use of an iterative approach.
- Recursion can be used to traverse String, array, and ArrayList objects.

EXCLUSION STATEMENT: writing recursive program code is outside the scope of the course and AP Exam.

sequential vs. Binary

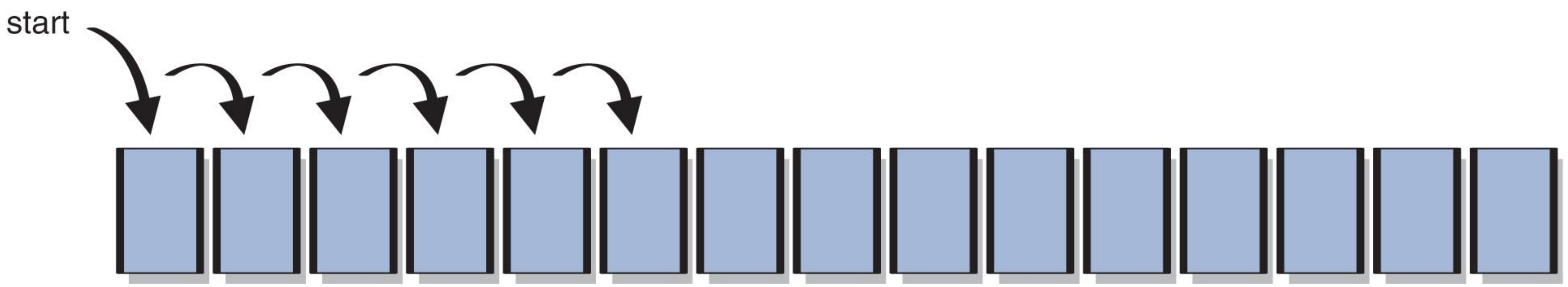





Binary search
steps: 0
37
1 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
low
mid
high
Sequential search
steps: 0
37
1 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
www.mathwarehouse.com

linear or sequential search

- A common task when working with arrays is to search an array for a particular element
- A linear or sequential search examines each element of the array in turn until the desired element is found



- See `Guests.java`

Note:Sequential or linear search is the only method that can be used to find a value in unsorted data.

Searching an array of `int`

```
public int whereIsMyNumber(int magicNumber, int [] myNumbers)
{
    for (int index = 0; index < myNumbers.length; index++)
    {
        if (myNumbers[index] == magicNumber)
        {
            return index;
        }
    }
    return -1;
}
```

Searching linear structure

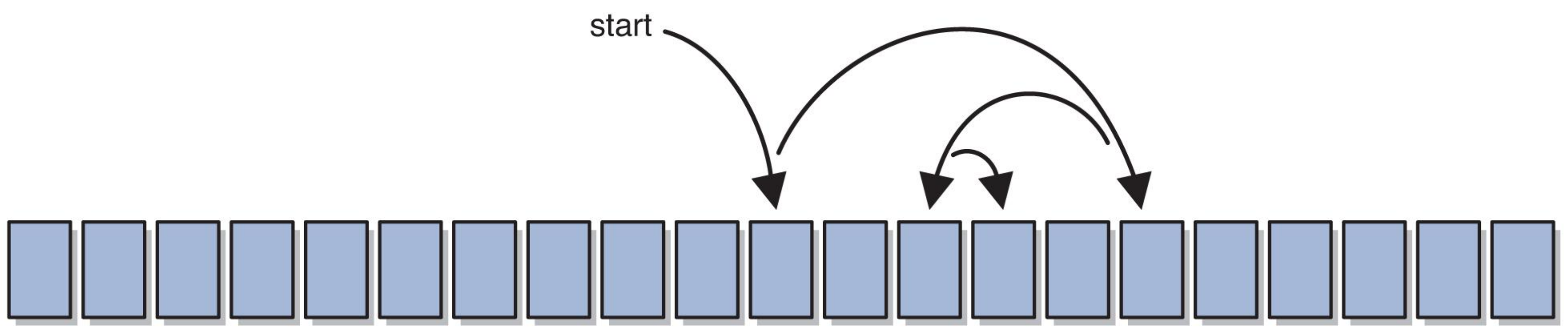
Finding information with a computer is something we need to know how to do. Linear search algorithms are BEST used when we do not have any idea about the order of the data and so we need to look at each element to determine if what we are looking for is in fact inside the array or **`ArrayList`**.

When searching, we do need to remember that different data types require different comparisons!

- When looking at **`int`** values, the == operator is the tool to use!
- When searching for a **`double`** value, we need to make sure the value is close enough by doing some math!
- **`Object`** instances should always use the **`.equals(otherThing)`** method to check for a match!

Binary search

- A binary search is more efficient than a linear search but it can only be performed on an **sorted** list
- A binary search examines the middle element and moves left if the desired element is less than the middle, and right if the desired element is greater
- This process repeats until the desired element is found



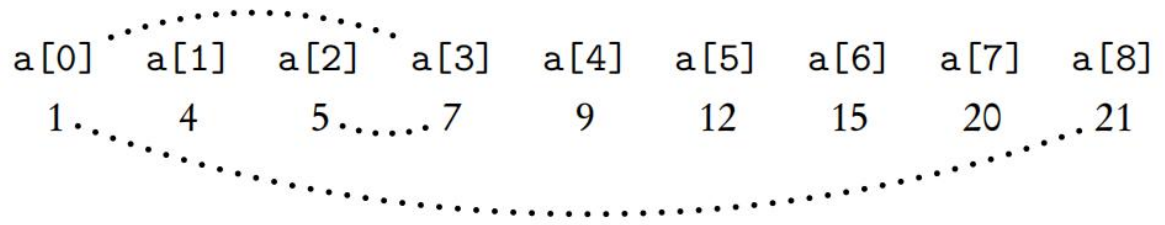
- Implement a binary search algorithm

```
public int binarySearch(int[] elements, int target) {
     int left = 0;
     int right = elements.length - 1;
     while (left <= right)
     {
        int middle = (left + right) / 2;
        if (target < elements[middle])
        {
           right = middle - 1;
        }
        else if (target > elements[middle])
        {
           left = middle + 1;
        }
        else {
           return middle;
        }
      }
      return -1;
   }
```

NOTE

When `low` and `high` cross, there are no more elements to examine, and `key` is not in the array.

Example: suppose 5 is the key to be found in the following array:

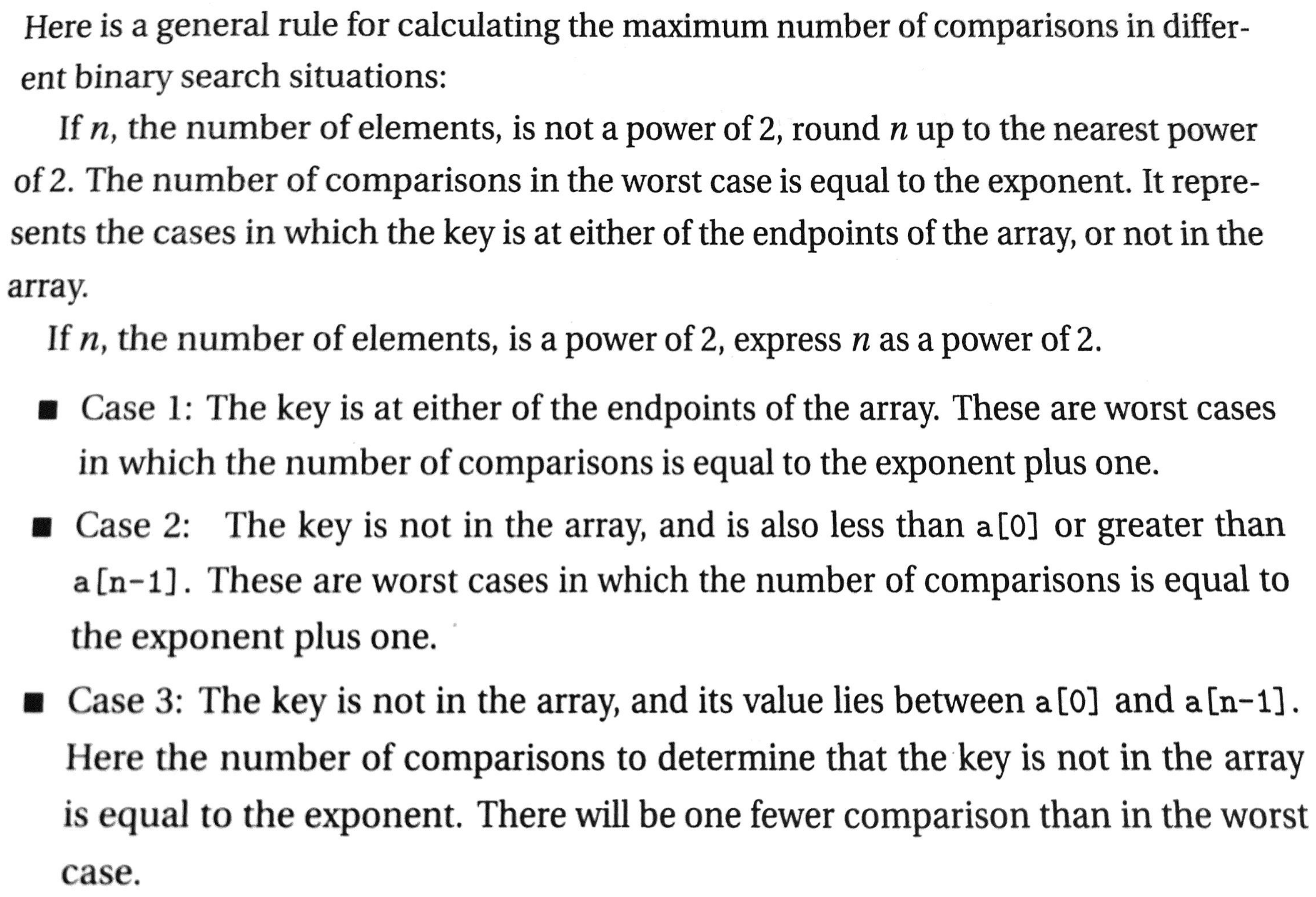


First pass:	`mid = (8+0)/2 = 4.`	Check `a[4]`.
Second pass:	`mid = (0+3)/2 = 1.`	Check `a[1]`.
Third pass:	`mid = (2+3)/2 = 2.`	Check `a[2]`. Yes! Key is found.

Analysis of Binary Search:

1. In the best case, the key is found on the first try (i.e., `(low + high)/2` is the index of `key`).
2. In the worst case, the key is not in the list or is at either end of a sublist. Here the n elements must be divided by 2 until there is just one element, and then that last element must be tested. An easy way to find the number of comparisons in the worst case is to round n up to the next power of 2 and take the exponent. For example, in the array above, $n = 9$. Suppose 21 were the key. Round 9 up to 16, which equals 2^4. Thus you would need four comparisons to find it. Try it!

Analysis of Binary Search



Here is a general rule for calculating the maximum number of comparisons in different binary search situations:

If *n*, the number of elements, is not a power of 2, round *n* up to the nearest power of 2. The number of comparisons in the worst case is equal to the exponent. It represents the cases in which the key is at either of the endpoints of the array, or not in the array.

If *n*, the number of elements, is a power of 2, express *n* as a power of 2.

- Case 1: The key is at either of the endpoints of the array. These are worst cases in which the number of comparisons is equal to the exponent plus one.
- Case 2: The key is not in the array, and is also less than `a[0]` or greater than `a[n-1]`. These are worst cases in which the number of comparisons is equal to the exponent plus one.
- Case 3: The key is not in the array, and its value lies between `a[0]` and `a[n-1]`. Here the number of comparisons to determine that the key is not in the array is equal to the exponent. There will be one fewer comparison than in the worst case.

Recursive Binary searching

Binary search

(recursive)

- ✓ Let’ s write a recursive version of Binary Search. Note that you can write solutions to many problems using recursion or iteration.
- ✓ Iteration is usually preferred and more efficient, but recursive solutions can be elegant and require less code.

- Thinking about
- 1.what is the base case?
    - What’s the base case for a recursive version of Binary Search (where we want the recursion to stop)? Remember that in binary search, we always check the middle element first when looking for a target element from a startIndex to an endIndex.
- 2.what is the non-base case?
    - Given a recursive binary search method with the method signature “`boolean binarySearch(int[] array, int target, int startIndex, int endIndex)`”, what recursive method call would search the array from index 0 to the middle index?

```
public static int recursiveBinarySearch(int[] array, int start, int end,
int target)
   {
       int middle = (start + end)/2;
       if (target == array[middle]) {// base case: check middle element
           return middle;
       }
       if(end < start){// base case: check if we've run out of elements
           return -1; // not found
       }

       if (target < array[middle]){// recursive call: search start to middle
           return recursiveBinarySearch(array, start, middle - 1, target);
       }

       if (target > array[middle]){// recursive call: search middle to end
           return recursiveBinarySearch(array, middle + 1, end, target);
       }
       return -1;
   }
```

Comparing search

- Informal run-time comparisons of program code segments can be made using **statement execution counts**.
- Binary search is much **faster** than linear search, especially on large data sets, but it can only be used on **sorted** data.

Sorting

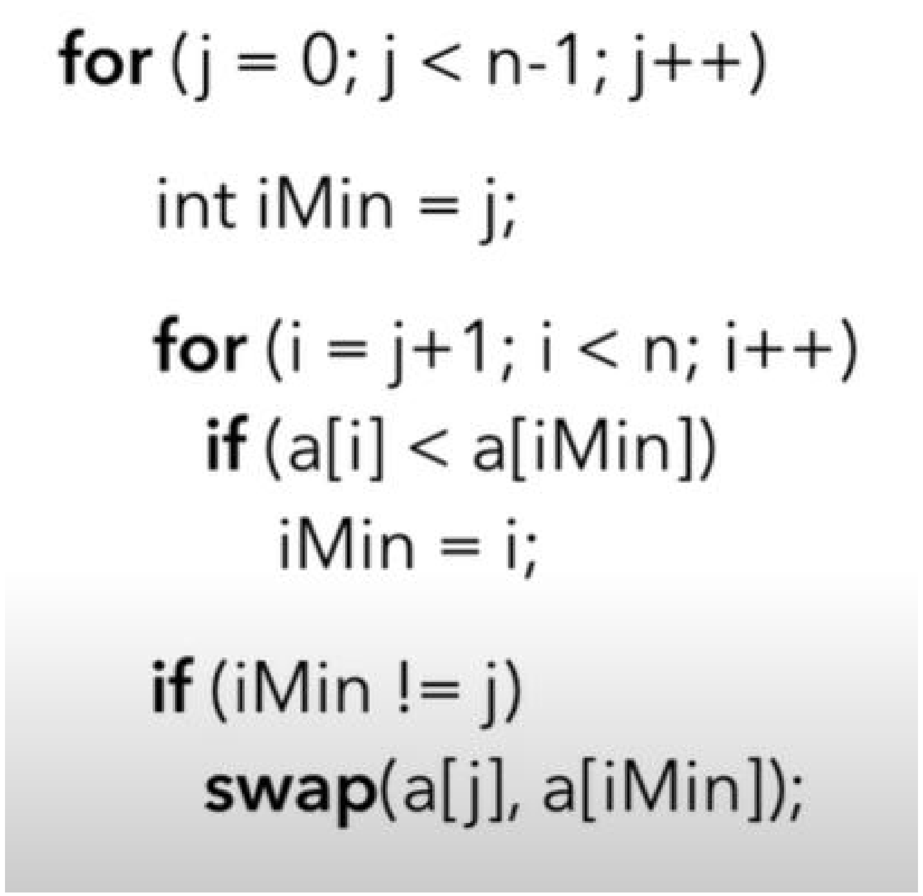
- There are many sorting algorithms to put an array or ArrayList elements in alphabetic or numerical order.
- The three sorting algorithms that you need to know for the AP CS A exam are:
    - **Selection Sort** - Select the smallest item from the current location on to the end of the array and swap it with the value at the current position. Do this from index 0 to the array length - 2. You don’ t have to process the last element in the array, it will already be sorted when you compare the prior element to the last element.
    - **Insertion Sort** - Insert the next unsorted element in the already sorted part of the array by moving larger values to the right. Start at index 1 and loop through the entire array.
    - **Merge sort** - Break the elements into two parts and recursively sort each part. An array of one item is sorted (base case). Then merge the two sorted arrays into one.

Selection sort

- The approach of Selection Sort:
  - select a value and put it in its final place into the list
  - repeat for all other values
- In more detail:
  - find the smallest value in the list
  - switch it with the value in the first position
  - find the next smallest value in the list
  - switch it with the value in the second position
  - repeat until all values are in their proper places

Selection sort

- pseudocode



```
for (j = 0; j < n-1; j++)

    int iMin = j;

    for (i = j+1; i < n; i++)
      if (a[i] < a[iMin])
        iMin = i;

    if (iMin != j)
      swap(a[j], a[iMin]);
```

java code

```
public static void selectionSort(int[] elements)
{
   for (int j = 0; j < elements.length - 1; j++)
   {
      int minIndex = j;
      for (int i = j+1; i < elements.length; i++)
      {
         if (elements[i] < elements[minIndex])
         {
            minIndex = i;
         }
      }
      if(minIndex!=j)
      {
          int temp = elements[j];
          elements[j] = elements[minIndex];
          elements[minIndex] = temp;
      }
    }
}
```

Insertion sort

- The approach of Insertion Sort:
    - pick any item and insert it into its proper place in a sorted sublist
    - repeat until all items have been inserted
- In more detail:
    - consider the first item to be a sorted sublist (of one item)
    - insert the second item into the sorted sublist, shifting the first item as needed to make room to insert the new addition
    - insert the third item into the sorted sublist (of two items), shifting items as necessary
    - repeat until all values are inserted into their proper positions

Insertion sort

- java code

```
public static void insertionSort(int[] elements)
   {
      for (int i = 1; i < elements.length; i++)
      {
         int temp = elements[i];
         int possibleIndex = i;
         while (possibleIndex > 0 && temp < elements[possibleIndex - 1])
         {
            elements[possibleIndex] = elements[possibleIndex - 1];
            possibleIndex--;
         }
         elements[possibleIndex] = temp;
      }
  }
```

pseudocode

```
for (i=1;i<n-1,i++)
   temp=a[i]
    j=i;
   while(j>0 and a[j-1]>a[j])
        a[j]=a[j-1]
         j--
    a[j]=temp
```

Merge Sort

- Selection and insertion sorts are inefficient for large n, requiring approximately n passes through a list of n elements. More efficient algorithms can be devised using a “**divide-and-conquer**” approach, which is used in both the sorting algorithms that follow.
- Here is a recursive description of how mergesort works:

✓If there is more than one element in the array

  ✓Break the array into two halves.

  ✓Mergesort the left half.

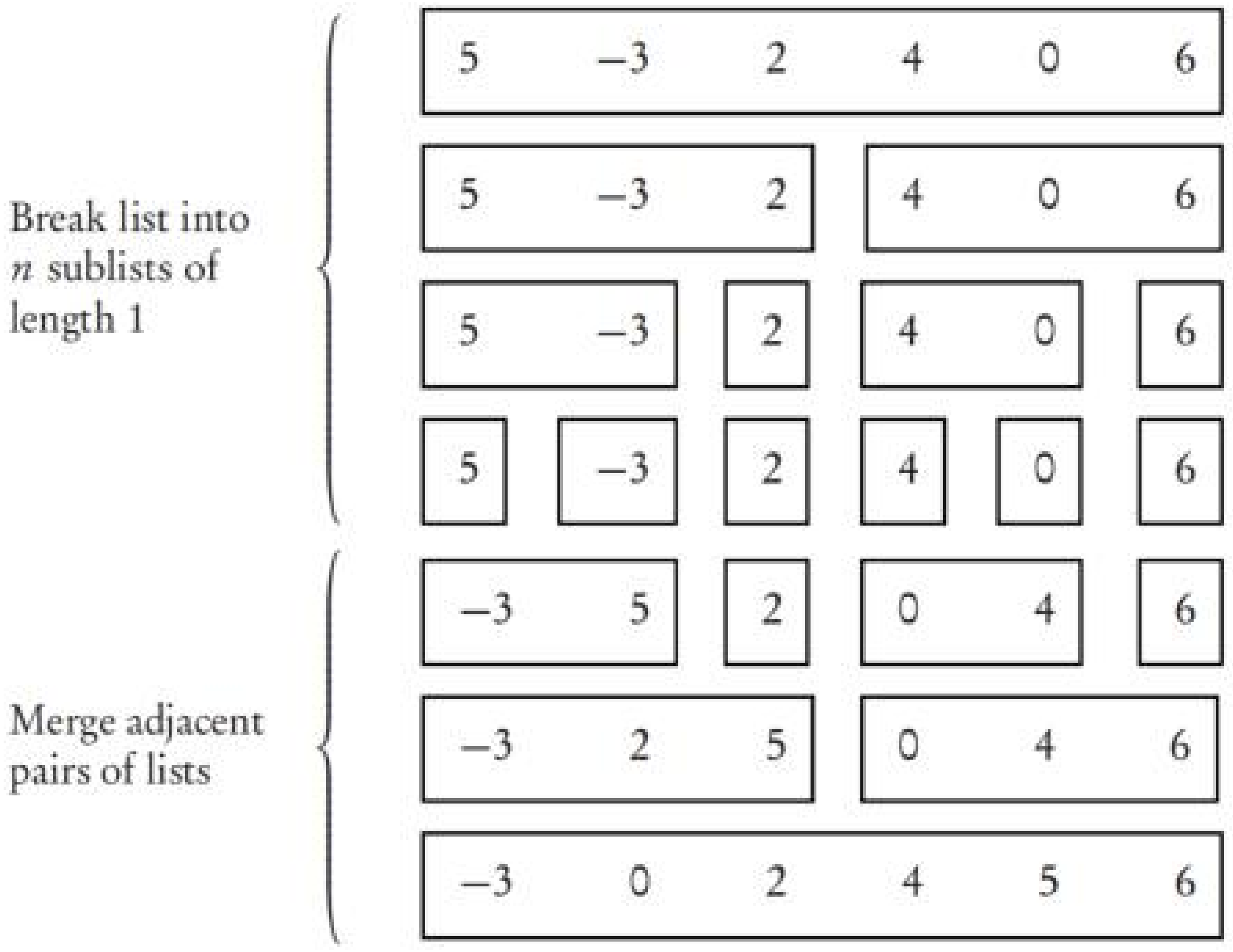
  ✓Mergesort the right half.

  ✓Merge the two subarrays into a sorted array

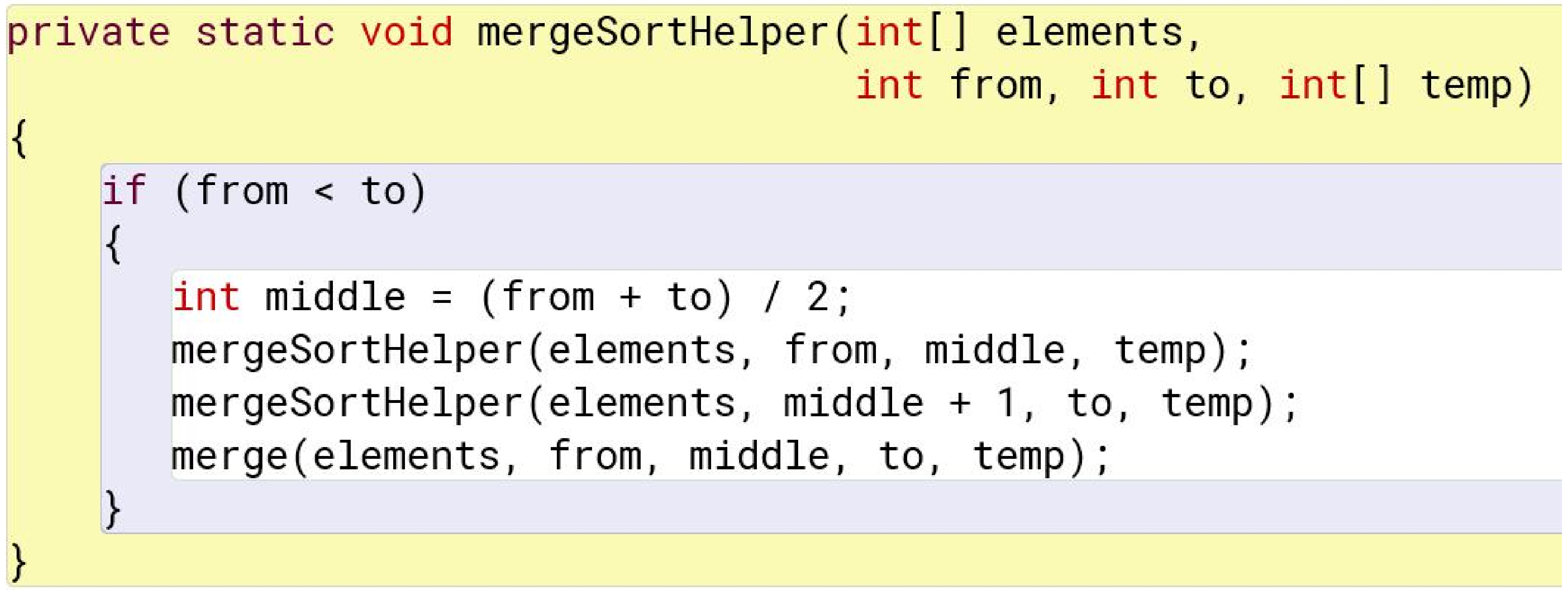
Merge Sort

An example of mergesort follows:

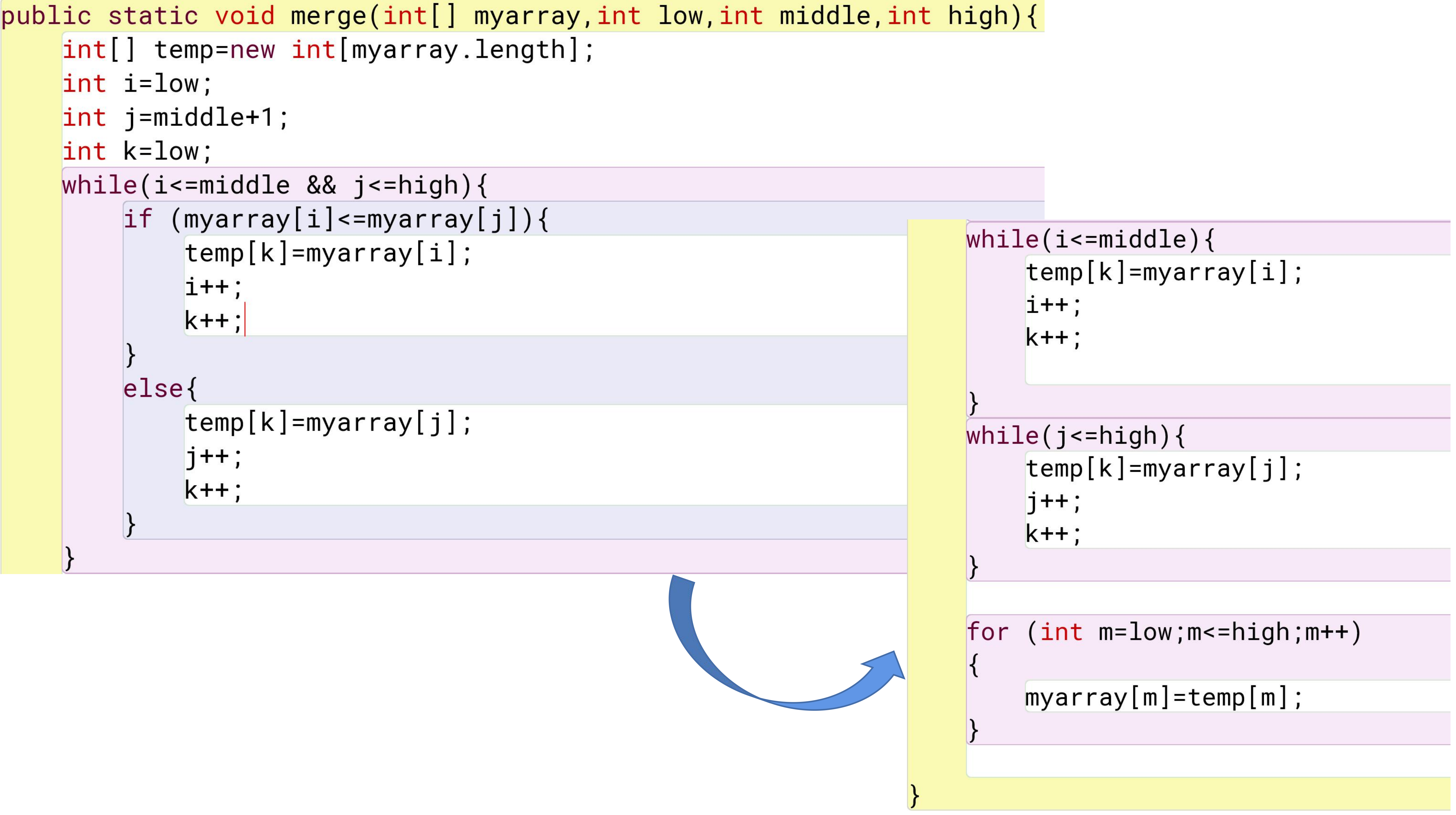
- ✓ Here' s what happens in mergesort:
  1. Start with an unsorted list of n elements.
  2. The recursive calls break the list into n sublists, each of length 1. Note that these n arrays, each containing just one element, are sorted!
  3. Recursively merge adjacent pairs of lists. There are then approximately n/2 lists of length 2; then, approximately n/4 lists of approximate length 4, and so on, until there is just one list of length n.



Merge Sort



```
private static void mergeSortHelper(int[] elements,
                                    int from, int to, int[] temp)
{
    if (from < to)
    {
        int middle = (from + to) / 2;
        mergeSortHelper(elements, from, middle, temp);
        mergeSortHelper(elements, middle + 1, to, temp);
        merge(elements, from, middle, to, temp);
    }
}
```



```
public static void merge(int[] myarray,int low,int middle,int high){
    int[] temp=new int[myarray.length];
    int i=low;
    int j=middle+1;
    int k=low;
    while(i<=middle && j<=high){
        if (myarray[i]<=myarray[j]){
            temp[k]=myarray[i];
            i++;
            k++;
        }
        else{
            temp[k]=myarray[j];
            j++;
            k++;
        }
    }
```

```
    while(i<=middle){
        temp[k]=myarray[i];
        i++;
        k++;

    }
    while(j<=high){
        temp[k]=myarray[j];
        j++;
        k++;
    }

    for (int m=low;m<=high;m++)
    {
        myarray[m]=temp[m];
    }

}
```

Comparing Sorts

- Both Selection and Insertion sorts are similar in efficiency
- They both have outer loops that scan all elements, and inner loops that compare the value of the outer loop with almost all values in the list
- Approximately n^2 number of comparisons are made to sort a list of size n
- We therefore say that these sorts have efficiency $O(n^2)$, or are of order n^2
- Merge sorts are more efficient: $O(n \log n)$