2023-04-07

Unit 10 Review

Recursive Terminology

Recursion	Recursive Programming
The definition of an operation in terms of itself. Solving a problem using recursion depends on solving smaller occurrences of the same problem.	Writing methods that call themselves to solve problems recursively. ● An equally powerful alternative to iteration (for, while loops, etc.) ● Particularly well-suited for solving certain types of problems.

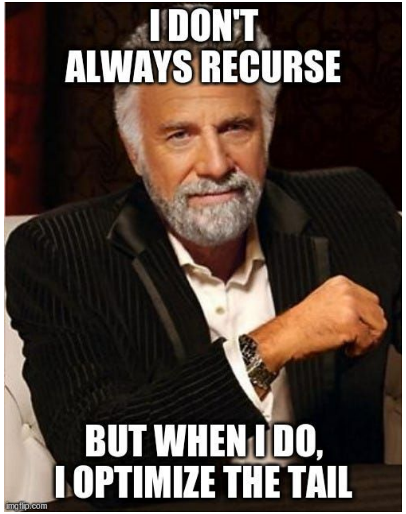
Recursive Terminology

Base Case	**Recursive Case**
The **base case** of a recursive method is the case where it does not recursively call itself, that is, the method terminates. The base case is a problem that is so simple, we already know the answer to it!	The **recursive case**, or **general case**, is the case where the method calls itself. It's called the general case because it's the case that usually happens when a recursive algorithm is executing. For the algorithm to work, the recursive case must diminish the problem so that it eventually approaches the base case.

Some recursive algorithms have more than one base or recursive case, but all have at least one of each.
A crucial part of recursive programming is identifying these cases.

Head Recursion vs. Tail Recursion

Head Recursion	Tail Recursion
If a method makes a recursive call to itself, but then does anything after that point, it is **head recursive**. (Sometimes the term used is **non-tail recursive**.)	If the recursive call is the last thing the method does, then the method is said to be **tail recursive**. Some languages (but not Java) can optimize out tail recursion automatically, turning it into a loop. This makes the method more efficient, and eliminates the risk of a stack overflow.



Tracing Recursive Methods

On the Unit 10 Test, and on the AP Exam, you may be asked to trace through "mystery" recursive methods and determine what is returned or output.

This isn't as simple as tracing through loops, where you can build a trace table, because a recursive method may call itself and then do additional manipulation on the result.

What you can do is draw a box every time you encounter a recursive call, and inside that box, determine the result of that recursive call.

(This may require drawing more boxes inside the boxes, recursively...)

Recursive Tracing

Consider the following recursive method.

```
public static int mystery(int n) {
  if (n < 10) {
    return n;
  } else {
    int a = n / 10;
    int b = n % 10;
    return mystery(a + b);
  }
}
```

What is the result of `mystery(648)`?

A recursive trace

This is essentially a call stack, just a different way of visualizing it!

```java
public static int mystery(int n) {
  if (n < 10) {
    return n;
  } else {
    int a = n / 10;
    int b = n % 10;
    return mystery(a + b);
  }
}
```

```
mystery(648):

 -  int a = 648 / 10;            // 64
 -  int b = 648 % 10;            //  8
 -  return mystery(a + b);       // mystery(72)

    mystery(72):

     -  int a = 72 / 10;          // 7
     -  int b = 72 % 10;          // 2
     -  return mystery(a + b);    // mystery(9)

        mystery(9):

         -  return 9;
```

Alternative solution for tail recursion

`mystery` is tail recursive. The recursive case is a single call to itself, and does no additional manipulation on the result. So, it can be converted to iteration.

```
public static int mystery(int n) {
  if (n < 10) {
    return n;
  } else {
    int a = n / 10;
    int b = n % 10;
    return mystery(a + b);
  }
}
```

→

```
public static int mystery(int n) {
  while (n >= 10) {
    int a = n / 10;
    int b = n % 10;
    n = a + b;
  }
  return n;
}
```

n	a	b
648	64	8
72	7	2
9		

Recursive Tracing with head recursion

Consider the following recursive method.

```
public static int mystery(int n) {
  if (n < 10) {
    return (10 * n) + n;
  } else {
    int a = mystery(n / 10);
    int b = mystery(n % 10);
    return (100 * a) + b;
  }
}
```

What is the result of `mystery(348)`?

Recursive Trace

```
public static int mystery(int n) {
  if (n < 10) {
    return (10 * n) + n;
  } else {
    int a = mystery(n / 10);
    int b = mystery(n % 10);
    return (100 * a) + b;
  }
}
```

```
mystery(348):

 -  int a = mystery(34);
      -  int a = mystery(3);
           -  return (10 * 3) + 3;     // 33
      -  int b = mystery(4);
           -  return (10 * 4) + 4;     // 44
      -  return (100 * 33) + 44;       // 3344
 -  int b = mystery(8);
      -  return (10 * 8) + 8;          // 88
 -  return (100 * 3344) + 88;          // 334488
```

For each call to the following method, indicate what value is returned:

```
public int mystery1(int x, int y) {
    if (x < y) {
        return x;
    } else {
        return mystery1(x - y, y);
    }
}
```

`mystery1(6, 13)`	
`mystery1(14, 10)`	
`mystery1(37, 10)`	
`mystery1(8, 2)`	
`mystery1(50, 7)`	

For each call to the following method, indicate what value is returned:

```java
public int mystery1(int x, int y) {
    if (x < y) {
        return x;
    } else {
        return mystery1(x - y, y);
    }
}
```

#	question	your answer	result
1	mystery1(6, 13)	6	pass
2	mystery1(14, 10)	4	pass
3	mystery1(37, 10)	7	pass
4	mystery1(8, 2)	0	pass
5	mystery1(50, 7)	1	pass

For each call to the following method, indicate what value is returned:

```
public int mystery3(int n) {
    if (n < 0) {
        return -mystery3(-n);
    } else if (n < 10) {
        return n;
    } else {
        return mystery3(n / 10 + n % 10);
    }
}
```

mystery3(6)	
mystery3(17)	
mystery3(259)	
mystery3(977)	
mystery3(-479)	

For each call to the following method, indicate what value is returned:

```
public int mystery3(int n) {
    if (n < 0) {
        return -mystery3(-n);
    } else if (n < 10) {
        return n;
    } else {
        return mystery3(n / 10 + n % 10);
    }
}
```

#	question	your answer	result
1	mystery3(6)	6	pass
2	mystery3(17)	8	pass
3	mystery3(259)	7	pass
4	mystery3(977)	5	pass
5	mystery3(-479)	-2	pass

For each call to the following method, indicate what value is returned:

```
public int mystery5(int x, int y) {
    if (x < 0) {
        return -mystery5(-x, y);
    } else if (y < 0) {
        return -mystery5(x, -y);
    } else if (x == 0 && y == 0) {
        return 0;
    } else {
        return 100 * mystery5(x / 10, y / 10) + 10 * (x % 10) + y % 10;
    }
}
```

`mystery5(5, 7)`	
`mystery5(12, 9)`	
`mystery5(-7, 4)`	
`mystery5(-23, -48)`	
`mystery5(128, 343)`	

For each call to the following method, indicate what value is returned:

```
public int mystery5(int x, int y) {
    if (x < 0) {
        return -mystery5(-x, y);
    } else if (y < 0) {
        return -mystery5(x, -y);
    } else if (x == 0 && y == 0) {
        return 0;
    } else {
        return 100 * mystery5(x / 10, y / 10) + 10 * (x % 10) + y % 10;
    }
}
```

#	question	your answer	result
1	mystery5(5, 7)	57	pass
2	mystery5(12, 9)	1029	pass
3	mystery5(-7, 4)	-74	pass
4	mystery5(-23, -48)	2438	pass
5	mystery5(128, 343)	132483	pass

10.2.1
Recursive Binary Search

Review: Binary Search (iterative approach)

```java
public static <T extends Comparable<T>> int binarySearch(ArrayList<T> array, T target) {
  int low = 0, high = array.size() - 1;
  while (low <= high) {
    int middle = (low + high) / 2;
    int compareResult = array.get(middle).compareTo(target);
    if (compareResult == 0) {
      return middle;
    } else if (compareResult < 0) {
      low = middle + 1;
    } else {
      high = middle - 1;
    }
  }
  return -1;
}
```

Recursive Binary Search

```java
public static <T extends Comparable<T>> int binarySearchHelper(ArrayList<T> array, T target, int low, int high) {
  if (low > high) {
    return -1;
  }
  int middle = (low + high) / 2;
  int compareResult = array.get(middle).compareTo(target);
  if (compareResult == 0) {
    return middle;
  } else if (compareResult < 0) {
    return binarySearchHelper(array, target, middle+1, high);
  } else {
    return binarySearchHelper(array, target, low, middle-1);
  }
}

public static <T extends Comparable<T>> int recursiveBinarySearch(ArrayList<T> words, T target) {
  return binarySearchHelper(words, target, 0, words.size() - 1);
}
```

Binary Search: Iterative or Recursive?

- The recursive solution is ever-so-slightly slower due to method call overhead.
- However, you may find it easier to understand... or harder to understand! Some programmers love recursion, some don't.
- A binary search is unlikely to get deep enough to cause a stack overflow, so it's not a problem to use the recursive method.
- If you go into the source code for the Java standard library implementation of binary search, it is probably iterative.
- The iterative version of binary search is essentially the recursive version with tail call optimization manually applied!

10.2.2: Merge Sort

Merge Sort

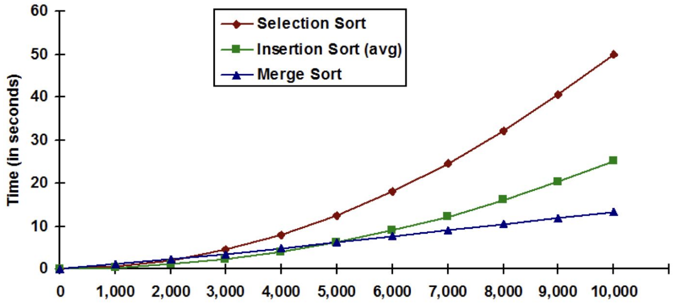
Merge Sort is a **divide and conquer** sorting algorithm, much like Binary Search is a divide and conquer search algorithm.

Merge Sort divides the input array in half, and recursively Merge Sorts the halves.

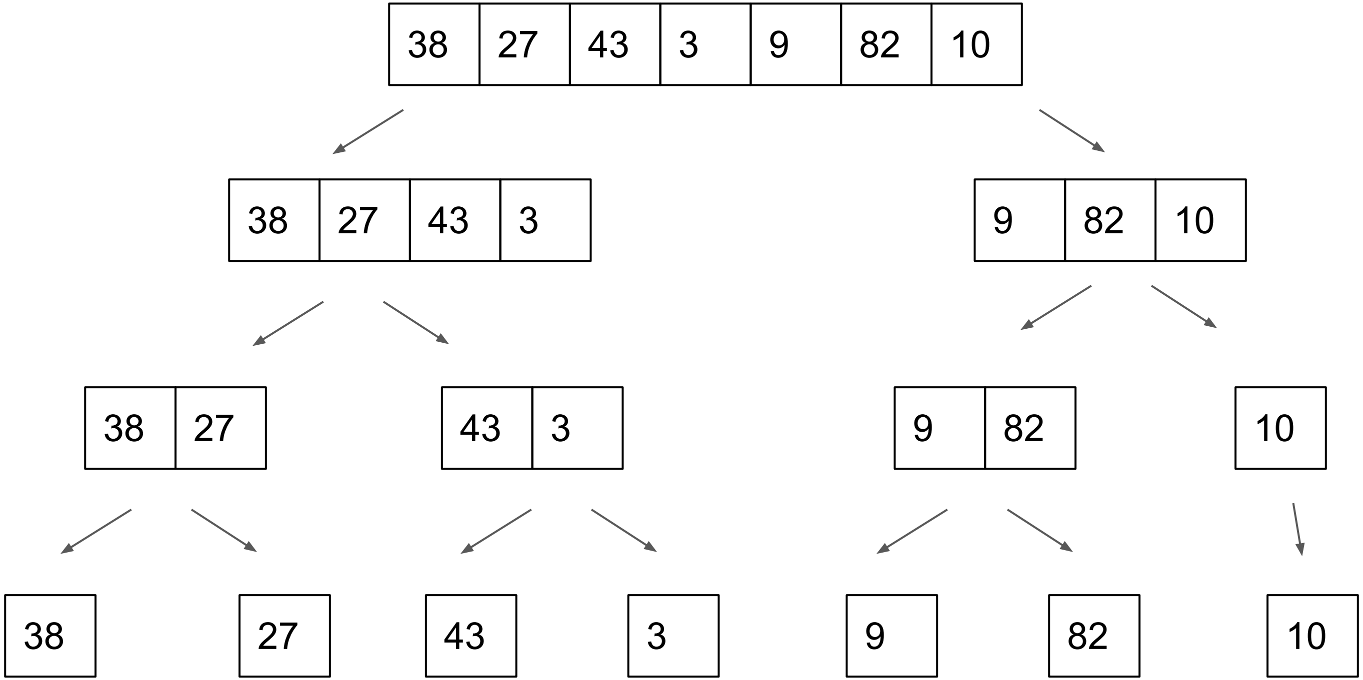
Merge Sort has O(N log N) running time for best case, worst case, and average case.

This is the lower bound for comparison-based sorting. It's proven that no algorithm has a superior worst-case running time.

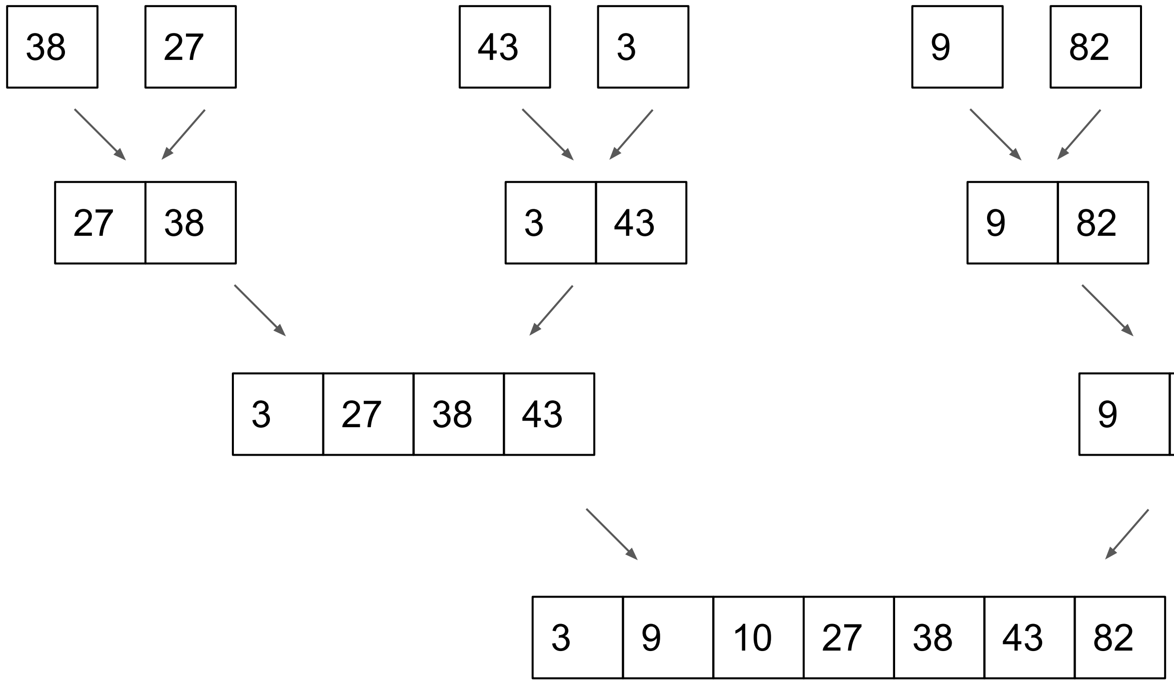
As typically implemented, Merge Sort requires O(N) temporary space.



Merge Sort: Split the array into halves down to one element



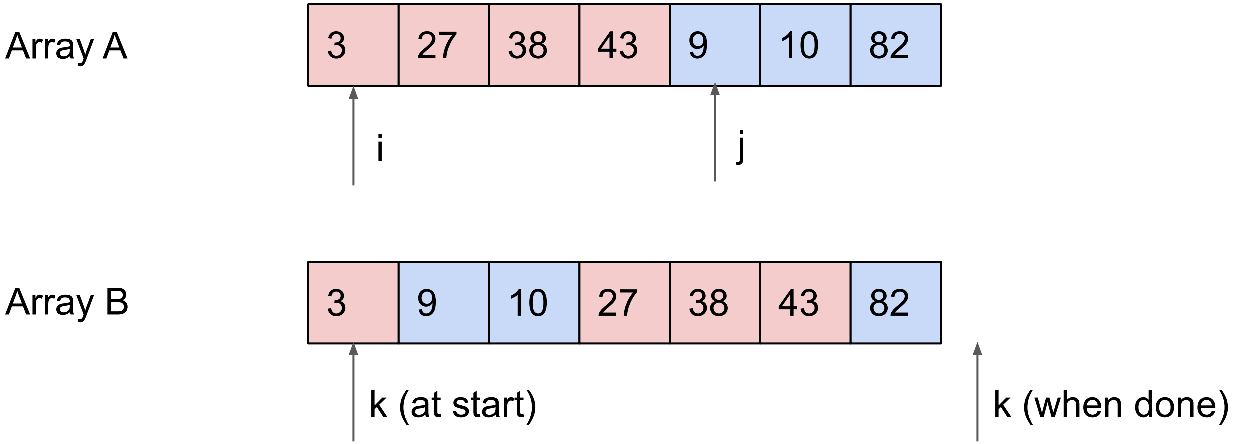
Merge Sort: Sort the halves and merge them together



Merge Sort Pseudocode

1. If the list’s size is 0 or 1, just return the original list (as it is sorted)
2. Split the list parameter int two lists of roughly equal size
3. Recursively Merge Sort both split lists, list 1 and list 2
4. Merge the two sorted lists and return the result

Merge Sort: The Merge



i = current index in left half, j = current index in right half. k = index in output array.

While i <= ending index of left half and j <= ending index of right half:
  B[k++] = smaller of A[i] and A[j].
  Advance i if we used A[i], or advance j if we used A[j].

After, one of the halves may still have something left... output anything left over into B.

CSAwesome's Merge Sort (edited for brevity) [1/2]

```
public static void mergeSort(int[] elements) {
   int n = elements.length;
   int[] temp = new int[n];
   mergeSortHelper(elements, 0, n - 1, temp);
}

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);
    }
}
```

CSAwesome's Merge Sort implementation [2/2]

```
private static void merge(int[] elements, int from, int mid, int to, int[] temp) {
   int i = from, j = mid + 1, k = from;

   while (i <= mid && j <= to) {
      if (elements[i] < elements[j]) {
         temp[k++] = elements[i++];
      } else {
         temp[k++] = elements[j++];
      }
   }

   while (i <= mid) {
      temp[k++] = elements[i++];
   }

   while (j <= to) {
      temp[k++] = elements[j++];
   }

   for (k = from; k <= to; k++) {
      elements[k] = temp[k];
   }
}
```

Here, we're comparing the current elements from each half, and taking the smaller one. We only advance the pointer for the side we took from!

If we got here, one or both halves are done. Output anything left over from the left half.

Output anything left over from the right half. (Only one of these loops will run.)

Copy everything back from the temp array. (This can be avoided with some extra fanciness.)

Acknowledgments

Some of the slides in this deck were adapted from

https://s3-us-west-2.amazonaws.com/www-cse-public/k12outreach/apcs/slides/java-recursive-tracing.pdf

(Slides provided by the University of Washington Computer Science & Engineering department. Adapted from slides by Marty Stepp, Stuart Reges & Allison Obourn.)

Some of the mystery problems were taken from https://practiceit.cs.washington.edu/