Answers and Explanations

Bullets mark each step in the process of arriving at the correct solution.

1. The answer is A.

   - Let's trace the calls. The parts in *italics* were filled in on the way back up. That is, the calls in the plain type were written top to bottom until the base case returned 1. Then the answers were filled in *bottom to top.*

     puzzle(10) = num + puzzle(num/2) = 10 + puzzle(5) = 10 + *8* = ***18***
     puzzle(5) = num + puzzle(num/2) = 5 + puzzle(2) = 5 + *3* = *8*
     puzzle(2) = num + puzzle(num/2) = 2 + puzzle(1) = 2 + *1* = *3*
     puzzle(1) = base case! return 1

   - Don't forget to do integer division.

2. The answer is E.

   - Let's trace the calls just like we did in problem 1.

     mystery(11) = 2 * 11 + mystery(9)
     mystery(9) = 2 * 9 + mystery(7)
     mystery(7) = 2 * 7 + mystery(5)
     mystery(5) = 2 * 5 + mystery(3)
     mystery(3) = 2 * 3 + mystery(1)
     mystery(1) = 2 * 1 + mystery(-1)

     Uh oh—the parameter skipped right over 0, and it's only going to get smaller. This example will recurse until there is a stack overflow error.

- You might be confused by the fact that sometimes people leave out else when writing code like this. Let's take a look at the code again:

```
if (k == 0)
    return 1;
return 2 * k + mystery(k - 2);
```

Why doesn't this have to say *else* return 2 * k + mystery(k − 2)? The purpose of an else clause is to tell the flow of control to skip that section when the if part is executed. So *either* the if clause or the else clause is executed. In this case, the if clause contains a return. Once a return is executed, nothing further in the method will be read and control returns to the calling method. We don't have to tell it to skip the else clause, because it has already gone off to execute code in another method. It doesn't matter whether you choose to write your code like this or to include the else, but don't be confused if you see it on the exam.

3. The answer is C.

- Here we go again!

enigma(9) = 3 + enigma(7) = 3 + *10* = ***13***
enigma(7) = 3 + enigma(5) = 3 + *7* = *10*
enigma(5) = 3 + enigma(3) = 3 + *4* = *7*
enigma(3) = 2 + enigma(2) = 2 + *2* = *4*
...the base case!
enigma(2) = 2...and up we go!

4. The answer is A.

- We don't really want to trace 15 calls of the `printStars` method, so let's see if we can just figure it after a few calls.

printStars(15): prints one * and calls printStars(14)
printStars(14): prints one * and calls printStars(13)
. . .
. . .
printStars(1): prints one * and returns (remember, void methods also return, they just don't return a value).

- Without tracing every single call, we can see the pattern. 15 stars will be printed.
- This recursive method is a little different because it is not a return method. That's allowed, but it is not very common.

5. The answer is C.

- This time with Strings!

weird("Hello") = weird("Hello" + "lo") = weird("Hellolo") = ***"Hellololololo"***
weird("Hellolo") = weird("Hellolo" + "lo") = weird("Hellololo") = *"Hellololololo"*
weird("Hellololo") = weird("Hellololo" + "lo") = weird("Hellolololo") = *"Hellololololo"*
. . . that has a length >10, so we just start returning s.

- This one is a little different because there is no computation on the "return trip."

6. The answer is D.

- The Binary Search algorithm looks at the element in the middle of the array, sees if it is the right answer, and then decides if the target item is higher or lower than that element. At that point, it knows which half of the array the target item is in. Then it looks at the middle element of that half, and so on.

- Here's our array:

1	2	3	4	5	6	7	8	9	10	11

- First pass: The middle element is 6. We are looking for 4. 4 < 6, so eliminate the right half of the array (and the 6). Now we are considering:

1	2	3	4	5	~~6~~	~~7~~	~~8~~	~~9~~	~~10~~	~~11~~

- Second pass: The middle element is 3. 4 > 3, so we eliminate the left half of the section we are considering (and the 3). Now we are considering:

~~1~~	~~2~~	~~3~~	4	5	~~6~~	~~7~~	~~8~~	~~9~~	~~10~~	~~11~~

- You can see that we will find the 4 on our next round, but the answer to the question is 4 5.
- Good to know: In this example, there always *was* a middle element when we needed one. If there is an even number of elements, the middle will be halfway in between two elements. The algorithm will just decide whether to round up or down in those cases (usually down, as integer division makes that easy).

7. The answer is C.

- Instead of tracing the code, this time we are going to reason out what the method is doing.
- Notice that factors is an overloaded method. Our first call has one `int` parameter, but the remaining calls all have three. Our first call, factors(10), will result in the call factors(10, 9, 0).
- Looking at the factors method with three parameters, we can see that each time through, we subtract one from check, and the base case is check == 1. We start with check = 9 and call the method eight more times before returning count. Count begins at 0 and will be incremented when number % check == 0. Since number is 10, and check will equal all the numbers between 1 and 9, that will happen twice, at 10 % 5 and at 10 % 2. When the return statement is reached, count = 2.

8. The answer is C.

- Option I is incorrect. factors(0) will call factors(0, -1, 0). Since check is decremented each time, we will move further and further from the base case of 1. Infinite recursion will result in a stack overflow exception.
- Option II is incorrect. factors(2) will call factors(2, 1, 0). Since check is decremented to 0 before checking for the base case, once again we have infinite recursion resulting in a stack overflow error.
- Option III is correct. factors(12, 2, 5) will increment count (12 % 2 == 0), decrement check to 1, and then check for the base case and return count.

9. The answer is B.

- Let's trace.

function(24, 3) = function(21, 4) + 2 = *19* + 2 = ***21***
function(21, 4) = function(17, 5) + 2 = *17* + 2 = *19*
function(17, 5) = function(12, 6) + 2 = *15* + 2 = *17*
function(12, 6) = function(6, 7) + 2 = *13* + 2 = *15*
function(6, 7) = 6 + 7 = 13 (base case!)

10. The answer is A.

- The `if` statement we are completing represents the base case and the recursive call.
- We need to keep going if sum > 9, so the base case, the case that says we are done, occurs at sum <= 9. That is the *condition*. If sum <= 9, all we need to do is return sum.

- If sum >= 9, we need to add the digits up again, but the question is, the digits of *what*? If you go back to the examples in the description, 999, for example, the `while` loop will add 9 + 9 + 9 and put the result in sum, which now = 27. Then the next step is to add 2 + 7. Since 27 is held in the variable sum, that's what we need to pass to the next round of recursion. The *argument* is sum.
- You don't need to understand the `while` loop to answer the question, but it is a pretty cool loop, so let's explain it here anyway.
  - dividend % 10 gives you the last digit of dividend
  - dividend / 10 gets rid of the last digit of dividend (because it is integer division)
  - Here's an example, using the number 365.

    365 / 10 = 36 remainder 5 (mod will give us 5, add it to sum)
    36 / 10 = 3 remainder 6 (mod will give us 6, add it to sum)
    3 / 10 = 0 remainder 3 (mod will give us 3, add it to sum)
    0 will cause the loop to terminate.

11. The answer is C.

- Binary Search works like this:
  - We look at the midpoint of a list, compare it to the element to be found (let's call it the key) and decide if our key is >, <, or = that midpoint element.
  - If our key = midpoint element, we have found our key in the list.
  - If our key > midpoint element, we want to reset the list so that we will search just the top half of the current list.
  - If our key < midpoint element, we want to reset the list so that we will search just the bottom half of the current list.
- Look at the given code. We can see those comparisons, and we can see the high and low ends of the list being changed to match the answers to those comparisons. This code implements Binary Search.