Chapter 15
Recursion

Lesson page 15-1. Recursion

Activity 15-1-1 Recursive definitions

Question 1. A recursive definition is a definition that defines something in terms of itself.

Question 2. A recursive method is a method whose body contains a call on itself (or whose body contains a call on another method that eventually ends up calling the first one).

Question 3. `A` is an ancestor of `P` if `A` is a parent of `P` or `A` is an ancestor of a parent of `P`.

Activity 15-1-2 First recursive method

Question 4. A recursive method has a base case, in which no recursive calls are used, and a recursive case, in which at least one recursive call is used.

Question 5. To "hide" the base case, we tested for the recursive case, as shown below. Thus, there is no explicit mention of the base case –either a test for it or a statement to be executed in the base case.

```
// Set elements of b[h..k] to 0
public static void setToZero(int[] b, int h, int k) {
    if (h <= k) {
        b[h]= 0;
        setToZero(b, h+1, k);
    }
}
```

We prefer not to write recursive methods this way. We believe it is better to make the base case explicit, as shown below:

```
// Set elements of b[h..k] to 0
public static void setToZero(int[] b, int h, int k) {
    if (h == k+1)
        return;
    b[h]= 0;
    setToZero(b, h+1, k);
}
```

Question 6. We generally write a recursive-procedure body in this form:

```
if (base case) {
    Process base case, without recursive calls
    return;
}
// { assertion that describes the recursive case }
Process recursive case
```

Question 7. We first state in English what the similar but smaller problem to be solved is (in the recursive case); in doing so, we try to write it in the form in which the specification of the method is written. Then, we simply have to write the name of the method followed by the parameter list, with the parameters replaced by the corresponding arguments. This rests on the notion that to find out what a call of any method does, make a copy of the method specification and replace occurrences of the parameters by the arguments of the call.

Activity 15-1-3 Second recursive method

Question 8. The easiest case to deal with is a `String` of length `0`.

Question 9. We include `p.charAt(0)` because that charcater is not a blank, so it should be included in the result.

Activity 15-1-4 Third recursive method

Question 10. Because `x = 1*x`, for any `x`, the product of `0` numbers is defined to be `1`. In the same way, because `x = 0+x`, for any `x`, the sum of `0` numbers is defined to be `0`. Also, because `b = true && b`, for any boolean `b`, the conjunction of `0` boolean values is defined to be **true**. Based on this idea, what do you think the disjunction of `0` boolean values is?

Question 11. We know that `n > 0` in the recursive case because of the assertion "`// {n > 0}`".

Question 12. This method takes advantage of the fact that `n >= 0`:

```
// = 0 + 1 + 2 + ... + n (for n>= 0)
public static int sum(int n) {
    if (n == 0)
        return 0;
    // { n > 0 }
    return sum(n-1) + n;
}
```

Activity 15-1-5 The recursive pattern

Question 13. The base case of a recursive method is the condition (based on the parameters of the method) under which the method does not call itself

recursively.

Question 14. The recursive case of a recursive method is the condition (based on the parameters of the method) under which the method calls itself recursively.

Question 15. The big idea behind writing a recursive method is, in the recursive case, to identify a problem that is similar to the specification of the method, but smaller.

Activity 15-1-6 Exercises on recursion

Question 16. Here's the filled-in function:

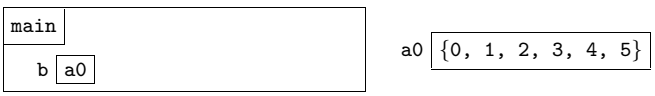
```
// = the largest value in b[h..k]
public static int largest(int[] b, int h, int k) {
    if (h == k)
        { return b[h]; }
    // {h < k, so b[h..k] has at least two values}
    int large= largest(b, h+1, k);
    if (b[h] > large) {
        return b[h];
    } else {
        return large;
    }
}
```

Lesson page 15-2. Execution of calls on recursive methods

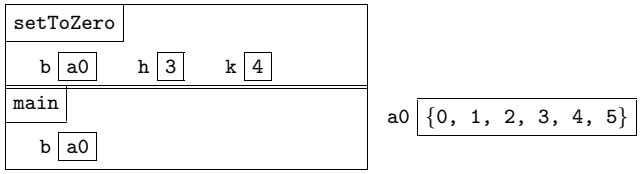
Activity 15-2-1 Execution of a call on a recursive method

Question 1. False. The model of execution given earlier works for recursive methods, without change.

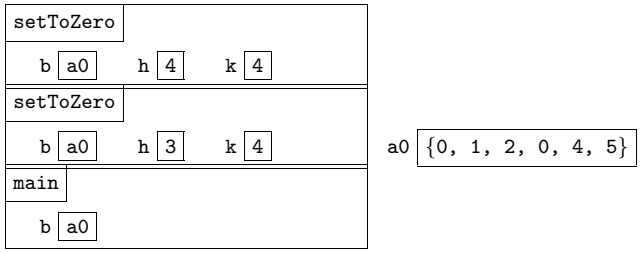
Question 2. We assume that these statements appear in method `main`, so that, after execution of the first statement, the call stack and array object are:



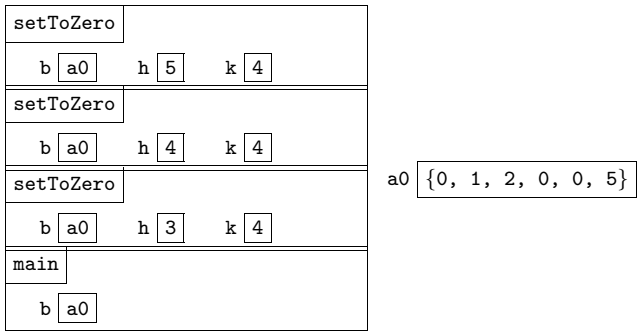
Snapshot when body of `setToZero` is about to be executed:



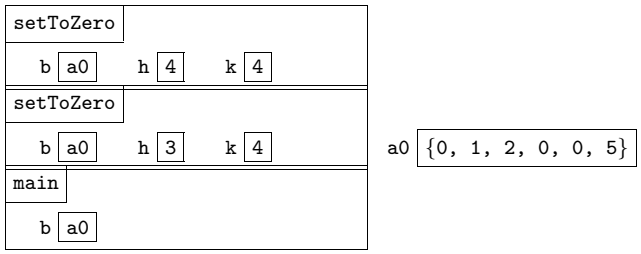
Snapshot when body of `setToZero` is about to be executed a second time:



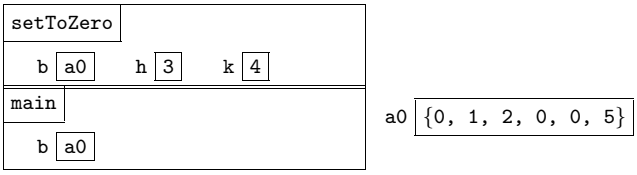
Snapshot when body of `setToZero` is about to be executed a third time:



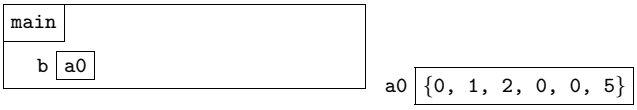
Snapshot upon termination of third call to `setToZero`:



Snapshot upon termination of second call to `setToZero`:



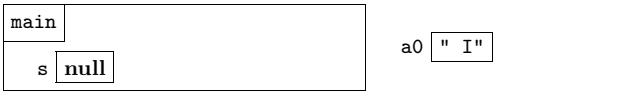
Snapshot upon termination of first call to `setToZero`:



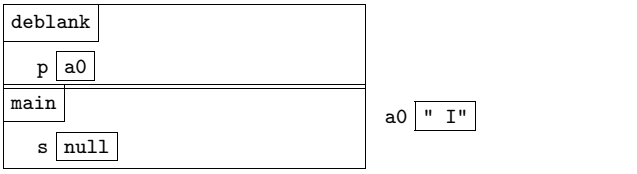
Question 3. We show execution of the following statement, which has a smaller argument in the call to `deblank`, in order to shorten the execution:

```
String s= deblank(" I");
```

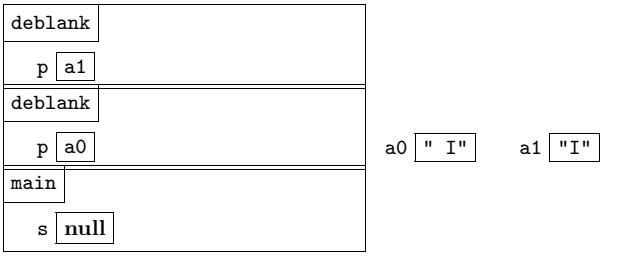
We assume that the statement appears in method `main`:



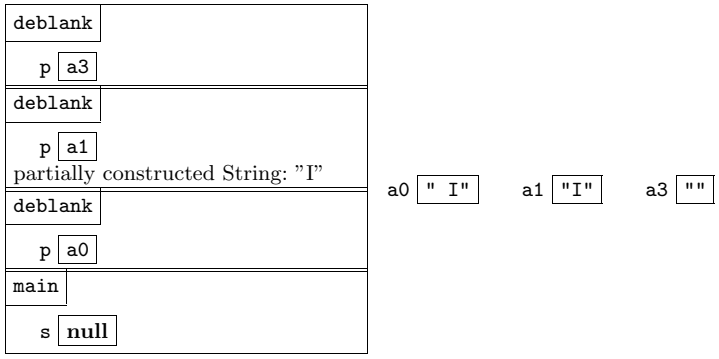
Snapshot when body of `deblank` is about to be executed:



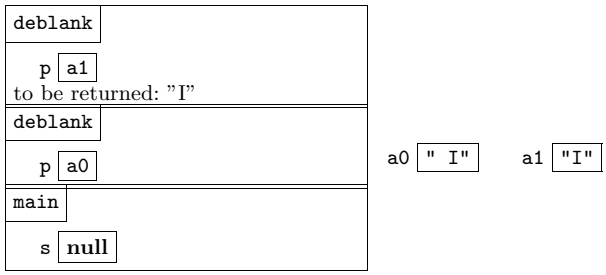
Snapshot when body of `deblank` is about to be executed a second time:



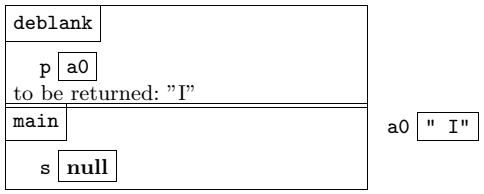
Snapshot when body of `deblank` is about to be executed a third time:



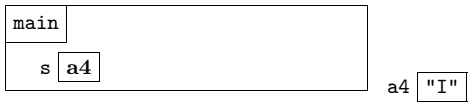
Snapshot upon termination of third call to `deblank`:



Snapshot upon termination of second call to `deblank`:



Snapshot upon termination of first call to `deblank`:



Activity 15-2-2 Tail recursion

Question 4. The depth of recursion is the number of calls on the method that have been made but have not yet completed.

Question 5. A recursive call is tail recursive if no other statement or expression is executed or evaluated after completion of the call —if the method body terminates immediately after the call.

Question 6. Method `fact1` does not have a tail-recursive call. the call `fact2(n-1,2*k)` in method `fact2` is tail recursive.

Activity 15-2-3 Implementation of tail recursion for procedures

Question 7. Here are the steps to transform a procedure body:

1. Insert a return statement at the end of the body, if there is not one there.
2. Place the procedure body in a while loop with loop condition **true**. To help the reader, label the loop with a label `tailRecursionLoop` and place a comment `// end tailRecursionLoop` just after the loop.
3. Replace each tail-recursive call by code that assigns the arguments to the parameters of the method and then terminates execution of the loop body (using **continue**).

Lesson page 15-3. Some interesting recursive methods

Activity 15-3-1 Tiling a kitchen

Activity 15-3-2 Computing x to the y

Activity 15-3-3 Mergesort

Lesson page 15-4. Quicksort

Activity 15-4-1 Basic quicksort

Question 1. An array with 10 or fewer elements generally makes a good base case for `quicksort`, and such arrays can be sorted using `insertionsort`.

Question 2. In the recursive case, the method partitions the array segment into three segments, based on a pivot value. The left segment has values that are at most the pivot value; the middle segment, equal to the pivot value; and the right segment, at least the pivot value. Then, the method sorts the left segment and sorts the right segment. The code is:

```
int j= partition(b,h,k);
quicksort(b,h,j-1);
quicksort(b,j+1,k);
```

Activity 15-4-2 Quicksort at its best

Question 3. Method `quicksort` works most efficiently when, at each partition step, the pivot value is the median value of the array segment.

Question 4. In the best case, `quicksort` makes at most `k * log(k)` swaps in sorting a segment of size `k`.

Question 5. In the best case, `quicksort` has at most `log(k)` frames for recursive calls allocated at any one time.

Activity 15-4-3 Quicksort at its worst

Question 6. Algorithm `quicksort` is least efficent when the pivot value used in each call on `partition` is the smallest or the largest value in the segment. This happens when the array is already sorted!

Question 7. In the worst case, `quicksort` makes up to `k*k` swaps in sorting a segment of size `k`.

Question 8. In the worst case, `quicksort` requires `k-1` frames for recursive calls when sorting a segment of size `k`.

Activity 15-4-4 Solving the quicksort space inefficiencies

Question 9. Before partitioning, permute the values of three array elements, usually the first, middle, and last ones, so that their median is in the first array element. This increases greatly the chance that the pivot value is closer to the median.

Question 10. Sort the smaller of the left and right segments first and remove tail-recursion, so that the larger segment is not sorted by a recursive call. Then, the depth of recursion, and hence the number of frames that are allocated at any time, is at most the logarithm of the size of the array segment.