Recursion termination: the bound function

Bound function

```
/** = 2^k.
  * Precondition k >= 0. */
public static int pow(int k) {
   if (k == 0) return 1;
   return 2 * pow(k-1);
}
```

Recursive function `pow` to the right has the base case `k = 0` and the recursive case `k > 0`. To show that a call like `pow`(20) terminates, we have to show that at some point the base case is reached. It's easy to see with this function because the recursive call `pow(k-1)` has an argument that is one less than parameter `k`, so the call `pow`(20) will result in the recursive call `pow`(19), which will result in the recursive call `pow`(18), etc., until the call `pow`(0) is executed.

Often, when we write recursive functions, as above, it is easy to see that they terminate. But sometimes termination can be trickier to show. Below, we use this example to give a general method for proving that a recursive method terminates.

```
/** Sort b[h..k] */
public static void QS(int[] b, int h, int k)
```

A recursive method may have a list of several parameters. Let's use `p` for the name of this list. For example, for function `pow` above, `p` is (`k`), and for procedure `QS` (for quicksort) whose header is shown to the right, `p` is (`b`, `h`, `k`). Similarly, we use `a` for the list of arguments of a recursive call

We formalize the notion of proving termination as follows.

Proof of termination of recursive calls of method f(p)

To prove termination of a call `f(p)`, exhibit a *bound function* `bf(p)` with the following properties:

1. For a base case `p`, `bf(p) ≤ 0`.
2. For a recursive case `p`, `bf(p) > 0`.
3. The arguments of recrusive call are "smaller" in the sense of bound function `bf` than the parameters of the method: `f(a)`, `bf(a) < bf(p)`.

Example 1. For function `pow` above, use the following bound function. You can easily check that the three properties are satisfied.

`bf(k) = k`.

Example 2. For procedure `QS` whose heading is given above, use

`bf(b, h, k) = k-h` i.e. (size of segment `b[h..k]`) `- 1`.

The base case will be when `b[h..k]` has 0 or 1 values, for then `b[h..k]` is already sorted. In that case `bf(b, h, k)` is `-1` or 0. If `b[h..k]` has 2 or more values, `bf(b, h, k)` > 1. To prove termination, then, we would have to show point 3 above: the segment to be sorted by each recursive call has to be smaller than segment `b[h..k]`.

```
/** = gcd(b, c).
  * Precondition 0 < b  and 0 < c. */
public static int gcd(int b, int c)) {
   if (b == c) return b;
   if (b > c) return gcd(b – c, c);
   return gcd(b, c – b);
}
```

Example 3. Function `gcd` to the right calculates the greatest common divisor of `b` and `c`. For example, `gcd`(5, 3) = 1 and `gcd`(9, 6) = 3. The function rests on these properties of `gcd` for `b` > 0 and `c` > 0:

(1) `gcd(b, b) = b`
(2) `gcd(b, c) = gcd(b, c–b) = gcd(b–c, c)`

We search for a bound function `bf(b, c)`. Looking at the two recursive calls, we think of using

`bf(b, c) = max(b, c)`

since if `b > c`, we have `max(b-c, c) < max(b, c)` and if `c > b`, we have `max(b, c-b) < max(b, c)`. That's good. However, in the base case, when `b = c`, we have `bf(b, c) = max(b, c) = b > 0`, while the properties for proving termination require `bf(b, c)` ≤ 0. Thus, we modify our choice of bound function to:

`bf(b, c) = max(b, c) – gcd(b, c)`

Discussion. This example shows that our rules for proving termination could be made more flexible. We could instead require that there exist a `k` such that for a base case, `bf(p) ≤ k`, and for a recursive case, `bf(p) > k`.

Recursion termination: the bound function

Example 4. To the right, function `isPal` determines whether its parameter `s` is a palindrome —whether `s` reads the same forward and backward. An obvious bound function `bf` is:

`bf(s)` = `s.length`() – 1

The base case is `s.length` () <= 1, and in this case, `bf(s)` <= 0. Second, in the recursive case, when `s` has at least 2 characters, `bf(s)` > 0. Third, the argument of the recursive call has 2 less chars in it than parameter `s`.

```
/** = "s is a palindrome" */
public static boolean isPal(String s) {
    if (s.length() <= 1) return true;

    // { s has at least 2 chars }
    int n= s.length()-1;
    return s.charAt(0) == s.charAt(n)  &&
            isPal(s.substring(1, n));
}
```

Example 5. We leave it to you to show that a suitable bound function for recursive function `fib`, to the right, is

`bf(n)` = `n - 1`

```
/** = fibonacci(n), for n >= 0 */
public static int fib(int n) {
  if (n <= 1) return n;
  // { 1 < n }
  return fib(n-2) + fib(n-1);
}
```