Exponentiation: Calculating b^n

Exponentiation
$b^0 = 1$.
For $n > 0$, $b^n = b * b^{n-1}$

The expression b^n stands for `b` raised to the power `n`. For natural numbers, —i.e. the nonnegative integers 0, 1, 2, …— b^n is defined recursively in the box to the right.

The second box on the right contains a recursive Java program for calculating b^n. It is simply a translation of the definition into Java. Parameter `b` has been given type **double** instead of **int** to make our discussion below a bit more reasonable. But the type of `b` could be anything, even **int**[] if we want to raise a matrix to a power.

```
/** = b^n.
  * Precondition: n ≥ 0. */
public static double
        pow(double b, int n) {
   if (n == 0) return 1;
   return b * pow(b, n–1);
}
```

This method obviously takes time O(`n`). It also takes space O(`n`) because each recursive call requires a frame for the call on the call stack. This is a real problem. In the JavaHyperText entry for exponentiation, we have a Java program that shows that attempting to calculate pow(.999, 16384) throws a stack-overflow exception —there is not enough space on the call stack.

A faster method based on an additional property

Often, the more properties we know of a function, the better program we can write. That is the case here. We use this property:

For even `n`, $b^n = (b*b)^{n/2}$

Example: $3^8 = (3*3) * (3*3) * (3*3) * (3*3) = (3*3)^4$

Method `powf` to the right incorporates this property.

```
/** = b^n.
  * Precondition: n ≥ 0. */
public static double powf(double b, int n) {
   if (n == 0) return 1;
   if (n%2 == 0) return powf(b*b, n/2);
   return b * powf(b, n–1);
}
```

Let us determine a bound on the depth of recursion for function `powf`. Suppose parameter `n` is 16. We have:

n = 16.	In binary: 10000
n –1 = 15.	In binary: 01111

A recursive call looks at the last bit of `n`:

- If that bit is 0, `n` is even, and the next recursive call use `n`/2 —that's `n` with the last bit thrown away.
- If that bit is 1, `n` is odd, and the next recursive call uses `n`–1 —that's `n` with its last bit changed from 1 to 0.

k	$n = 2^k$	n in binary	log n
0	1	0	0
1	2	10	1
2	4	100	2
3	8	1000	3
4	16	10000	4
5	32	100000	5

Therefore, the number of recursive calls is at most 2 times the number of bits needed to represent `n` in binary. The table to the right indicates that the number of bits needed to represent integer `n` in binary is 1+log `n`. Therefore, the depth of recursion is at most 2 + 2 log `n`, which is O(log `n`). Therefore, both the time and the space requirement of function `powf` is O(log `n`). Neat!

We urge you to download the Java program mentioned above and run it. The output of the program contains this information:

```
The call pow(.9999, 16384)    overflowed the call stack.
The call pow(.9999, 8192)     returned 0.4407660854530316.
The call powf(.9999, 8192)    returned 0.4407660854530856.  Recursion depth: 15
The call powf(.9999, 16384)   returned 0.19427474208563675. Recursion depth: 16
The call powf(.9999, 16383)   returned 0.194294171502787.   Recursion depth: 28
```

The marked difference between linear and log space becomes strikingly clear with this example. Function `pow` required linear space —8192 stack frames to calculate $.9999^{8192}$. Function `powf` required logarithmic space —only 15 stack frames!

(continued on next page)

Exponentiation: Calculating b^n

An iteratve version

```
// Given n ≥ 0, store b^n in z.
double x= b;
int y= n;
double z= 1.0;
// invariant: P
while (y != 0) {
   if (y%2 != 0) {z= z*x; y= y-1;}
   else {x= x*x; y= y/2;}
}
```

We can do better. Recursive function `powf` does take space O(log `n`). An iterative version of this function will take space O(1) —space is needed just for a few local variables. Further, though its running time remains the same as that of `powf` — O(log `n`)— it does not require *any* method calls and is thus faster. If time is at a premium and iteration provides a natural solution to a problem, it is better to use iteration rather than recursion

The other point about the iterative version is that it cannot be understood without the loop invariant.

We give the iterative algorithm in the box to the right. You can see that:

- If `y` is odd, it is decremented;
- If `y` is even, it is halved.
- The loop terminates when `y` = 0.

This is much like the recursive version. But why are `x`, `y`, and `z` initialized the way they are? Why the funny change to `x` and `z` in the loop body? And how do we know that `z` contains b^n upon termination? It's all a mystery.

This loop invariant will dispel the mystery:

`P: y ≥ 0` and $b^n = z*x^y$

We check the loopy questions.

1. Check that setting `z` = 1.0, `x` = `b`, and `y` = `n` makes invariant `P` true. It does.
2. If `y` = 0, then $x^y = 1$, and the invariant reduces to b^n = `z`. So `z` contains the correct value upon termination.
3. Each iteration reduces `y`, so termination is guaranteed.
4. Each iteration keeps invariant `P` true:
   a. If `y` is odd, execution of `z= z*x; y= y–1;` keeps the value of $z*x^y$ unchanged.
   b. If `y` is even, execution of `x= x*x; y= y/2;` keeps the value of x^y and thus $z*x^y$ unchanged.