A0.

```
>>> functools.reduce(lambda x, y: x * y, [1, 3, 5, 7, 9])
945
```

A1. [Here is an iterative solution and a recursive solution, next to each other.]

```
def reduce(f, l):
    if len(l) == 1:
        return l[0]
    else:
        acc = f(l[0], l[1])
        for i in range(2, len(l)):
            acc = f(acc, l[i])
        return acc
```

```
def reduce(f, l):
    if len(l) == 1:
        return l[0]
    else:
        return reduce(f, [f(l[0], l[1])] + l[2:])
```

B0. My kernel is `[[1, 2, 0], [2, 0, -2], [0, -2, -1]]`. Neighboring pixels above-left contribute positive amounts to the convolution. Neighboring pixels below-right contribute negatively. So the convolution result is large if the former are greater than the latter.

B1. For an $n \times n$ image convolved by a $k \times k$ kernel, the running time of `imageConvolved` is $\mathcal{O}(n^2k^2)$. For `imageConvolved` loops over (almost) all the rows and columns of the image, touching $\mathcal{O}(n^2)$ pixels. At each pixel, it constructs a weighted sum of k^2 nearby pixels. So the total time is proportional to n^2k^2.

[More precisely, the algorithm loops over $n - (k - 1)$ rows and $n - (k - 1)$ columns. But typically n is much greater than k, so $n - (k - 1)$ is roughly equal to n. More precisely still, the running time is proportional to $(n - (k - 1))^2k^2$, but this is $\mathcal{O}(n^2k^2)$, because after all $\mathcal{O}$ describes an upper bound on the running time.]

C0. [I'll omit the graph itself. It should show n on the horizontal axis and running time on the vertical axis. `euc` should plot as a straight line from the origin with positive slope. `gcd` should plot as an exponential curve, starting out below the `euc` line, but soon shooting dramatically above the `euc` line. I would label the n where they cross "N".]

`gcd`'s running time is $\mathcal{O}(2^n)$, while `euc`'s running time is $\mathcal{O}(n)$. At a particular value of n, which we might call N, the two curves cross. `euc` is faster than `gcd` for all $n > N$, and it's much faster when n is much greater than N.

[Some students said that trial division was $\mathcal{O}(n^2)$ rather than $\mathcal{O}(2^n)$. There is a huge difference. When $n = 64$, for example, $n^2 = 4{,}096$ and $2^n \approx 10^{19}$. When $n = 1{,}024$, as is common in RSA in 2016, $n^2 \approx 10^6$ and $2^n \approx 10^{308}$.]

C1. GCDs are important because they are a basic arithmetic computation. For example, we use them in elementary school when reducing fractions to their lowest terms (such as $60/24 = 5/2$). For another example, the RSA cryptosystem is a popular algorithm for protecting sensitive information such as financial transactions. And GCDs are needed for setting up RSA (in choosing the keys e and d).

The Euclidean algorithm is extremely, dramatically faster than the trial division algorithm. For example, when the numbers are 64 bits, `euc` uses roughly 64 operations while `gcd` uses roughly $2^{64} \approx 16{,}000{,}000{,}000{,}000{,}000{,}000$ operations. If the RSA setup used the trial division algorithm, then setting up RSA would be approximately as difficult as breaking RSA — rendering the whole algorithm pointless.

D.

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

E. When the command executes, it eventually forms a stack of recursive calls like this:

```
mergeSort(['1', 4])
mergeSort([3, 0, '1', 4])
mergeSort([5, 2, 7, 6, 3, 0, '1', 4])
```

The top-most call recursively determines that `['1']` and `[4]` are sorted. Then it tries to merge those two lists. The merging requires a comparison of the string `'1'` and the integer `4`, which causes an error. When Python reports this error, it shows the call stack, or at least which functions are on the call stack. In this case, the call stack looks like three copies of `mergeSort`.

[See below for an actual transcript. Remarkably, a couple of students came close to this level of detail in their answers.]

```
>>> mergeSort([5, 2, 7, 6, 3, 0, '1', 4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in mergeSort
  File "<stdin>", line 9, in mergeSort
  File "<stdin>", line 15, in mergeSort
TypeError: unorderable types: str() < int()
```

F0. Notice that Python's response at the end is a float, not an int.

```
>>> velveeta = Employee(1990, 60000)
>>> velveeta.payMonth()
>>> velveeta.getPayThisYear()
5000.0
```

F1. [This question is a bit open-ended. At a minimum, two lines must be added to `__init__` and two lines must be added to `payMonth`, and there must be some way to query the cumulative retirement account. The `getRetireMonthly` method and many other methods would be nice, but they're not strictly necessary.]

```
def __init__(self, birthYear, salary, retireMonthly):
    self.birthYear = birthYear
    self.salary = salary
    self.retireMonthly = retireMonthly
    self.retirement = 0
    self.startNewYear()

def payMonth(self):
    self.payThisYear += self.salary / 12.0
    self.payThisYear += self.retireMonthly
    self.retirement += self.retireMonthly

def getRetirement(self):
    return self.retirement

def getRetireMonthly(self):
    return self.retireMonthly
```