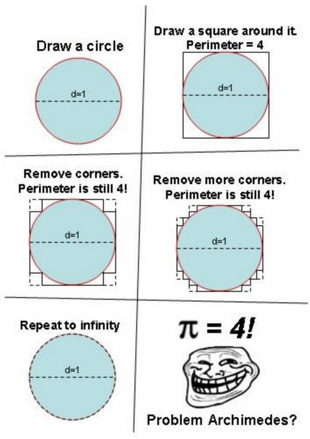
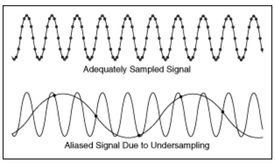
In [1]:
```
import numpy as np
```

Problem 1

The true perimeter of the lake is not correctly captured by a zoomed-out view because by doing so, you are assuming that the lake shore is smooth and does not have bends and folds, when in fact it does. I've seen a really helpful demonstration of the flaws in assuming that a densely folded object has the same perimeter as a smoothed approximation (source: http://joyreactor.com/post/285895):



This issue also brings to mind signal aliasing, which I learned about while working in optics/imaging as an undergraduate. Signal aliasing occurs when a waveform is sampled at a much lower rate than the waveform frequency, so a rapidly oscillating signal appears to be smooth (source: https://zone.ni.com/reference/en-XX/help/371361R-01/lvanlsconcepts/aliasing/):



To properly sample an underlying curve, the sampling rate has to be twice the frequency of the waveform. This is called the "Nyquist limit". While we are not dealing with sinusoidal functions in the case of the perimeter, the effect is similar in that the location of the coast rapidly fluctuates at a higher frequency than the size represented by a pixel on the zoomed out image.

In solving this problem, we have to consider what is practical in terms of measuring coastline for the purposes of dividing it amongst property owners. A property owner would not, for example, care if the rocks on the lakefront are mossy or not. At a cellular scale, there would be a great deal of perimeter involved that would have no bearing on the use of the lakefront. Therefore, I would define a minimum lengthscale at which measuring oscillations is meaningful. I would probably set this at 1 meter. Below 1 meter, any features will be the size of rocks, and probably won't be useful for building a dock, fishing, etc. I would therefore sample the image at 0.5 meter pixel size (the Nyquist limit for a 1 meter wavelength). I would use the difference in color between lake and land to fit a smooth curve to the pixel data, and would use the length of this curve to define the lake perimeter.

Problem 2

In [2]:
```
# Example input
day_i = ["Bob:100", "Alice:100", "Celia:110", "Bob:20"]
def purchase_summary(input):
    output = {}
    for entry in sorted(input):
        spl = entry.split(":")
        # Add value if key already exists; otherwise, set to value
        output[spl[0]] = output.get(spl[0], 0) + int(spl[1])
    return output
print(purchase_summary(day_i))
```

```
{'Alice': 100, 'Bob': 120, 'Celia': 110}
```

Problem 3

In [3]:
```
# Problem 3
def print_num(num):
        print("Your number is {:d}".format(num))
def play_number_guessing():
    # Add a few checks on user input so that the game won't continue upon bad values
    try:
        lower = int(input("What is the lower bound? "))
        upper = int(input("What is the upper bound? "))
    except:
        print("Error- bad bounds.")
        return
    if upper<lower:
        print("Error- bad bounds where upper<lower.")
        return
    # Continue the guessing cycle until guessing correctly and breaking out
    while True:
        # If we only have one number in range, that will be the correct guess
        if (upper-lower)==0:
            print_num(lower)
            break
        # Guess in the middle of the two values- use floor to arbitrarily split even ranges
        guess = int(np.floor((upper-lower)/2)+lower)
        correct = ''
        while correct != 'y' and correct != 'n':
            correct = input("Is {:d} your number? ('y' or 'n'): ".format(guess))
        if correct == 'y':
            print_num(guess)
            break
        else:
            # If we only have two numbers, we automatically know it is the one not guessed
            if upper-lower == 1:
                if guess == lower:
                    print_num(upper)
                else:
                    print_num(lower)
                break
            guess_high = ''
            # Determine whether the guess is higher than true number
            while guess_high != 'y' and guess_high != 'n':
                guess_high = input("Is your number less than {:d}? ('y' or 'n'): ".format(guess))
            # Bounds are inclusive, and we know guess is false, so set to guess+-1
            if guess_high == 'y':
                upper = guess-1
            else:
                lower = guess+1
    return
play_number_guessing()
```

```
What is the lower bound? 3
What is the upper bound? 20
Is 11 your number? ('y' or 'n'): n
Is your number less than 11? ('y' or 'n'): y
Is 6 your number? ('y' or 'n'): n
Is your number less than 6? ('y' or 'n'): n
Is 8 your number? ('y' or 'n'): n
Is your number less than 8? ('y' or 'n'): n
Is 9 your number? ('y' or 'n'): n
Your number is 10
```

Problem 4

I chose to watch Unsupervised Machine Learning Python. The following aspects matched with my learning style:

- I liked how the professor demonstrated the "curse of dimensionality" with the three different dimensional images. I find it helpful when a professor proves/shows a concept that they are teaching instead of just telling us that it is true, because this feels more sound and helps me remember the concept better. I really liked the simple demonstration of how the data became more sparse.
- I really liked the "toy model" of data in two dimensions and the way he showed how one dimension exhibited a larger range of data. I learn really well by coding and creating simulations, and seeing this toy model helped me think about how I might program the PCA in real life.
- I enjoyed the geometric demonstration of PCA and the way that he showed the orthogonal projections on the different axes. I like geometry a lot and learn well from geometric proofs and arguments. It is especially helpful when a professor uses a 3D geometry to demonstrate abstract linear algebra concepts like he did here.

For a suggestion, I think it might help to motivate some of the concepts more while defining them. It may be because I haven't watched the previous lectures, but I struggled a little with following the lecture before reaching the visual demonstrations. After that, I understood it much better. I think more focus on the big picture in the beginning could have helped.

In [ ]:
```
```