Assignment 1

BMI 2005 - BioAlgorithms
Due: 01/30/19

By: **Ryan Neff**
ryan.neff@icahn.mssm.edu

Question 1

Lennard Jones potential (LJ) function:

$$V_{lj} = 4 * \epsilon * ((\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6})$$

Where:

- episilon, e: depth of potential well (strength of non-bonded attraction between two atoms)
- sigma, s: distance at which inter-particle potential is 0
- distance, r: distance between particles

Question 1a

In [1]:
```
def naive_lennard_jones(r,s=1,e=1):
    '''naive_lennard_jones(r)
     Computes the potential energy V_ij between two atoms i and j
     with distance r where e and s are set to 1

     Inputs:
        r (float)
            Distance between two atoms i and j

    Outputs:
        V_ij (float)
            Potential energy between two atoms
    '''
    return 4*s*e*(s**12/r**12 - s**6/r**6)
```

Question 1b

```
efficient algorithm equation:
    V_ij = -4*e*u(1-u)
substitute z**3 for u:
    V_ij = -4*e*z^3(1-z^3)
substitute s^2/r^2 for z:
    V_ij = -4*e*(s^2/r^2)^3(1-(s^2/r^2)^3)
distribute powers:
    V_ij = -4*e*(s^6/r^6)(1-(s^6/r^6))
move s^6/r^6 to inside parenthesis:
    V_ij = -4*e*((s^6/r^6)*1 - (s^6/r^6)*(s^6/r^6))
distribute powers:
    V_ij = -4*e*((s^6/r^6) - (s^12/r^12))
move -1 to inside parenthesis:
    V_ij = 4*e*(-(s^6/r^6) + (s^12/r^12))
    V_ij = 4*e*((s^12/r^12)-(s^6/r^6))
Let x^z/y^z = (x/y)^z.
    therefore:
        V_ij = 4*e*((s/r)^12-(s/r)^6)
***This is the original equation. QED***
```

Question 1c

In [2]:
```
def efficient_lennard_jones(r,s,e):
    '''efficient_lennard_jones(r,s,e)
     Computes the potential energy V_ij between two atoms i and j

     Inputs:
        r (float)
            Distance between two atoms i and j
        s (float)
            Sigma, distance at which inter-particle potential is 0
        e (float)
            Epsilon, strength of non-bonded attraction between two atoms

    Outputs:
        V_ij (float)
            Potential energy between two atoms
    '''

    r2 = r**2
    s2 = s**2
    z = s2/r2
    u = z**3
    return -4*e*u*(1-u)
```

Performance Analysis

```
naive:
    code:
        return 4*s*e*(s**12/r**12 - s**6/r**6)

    First step: r**12
    Second step: s**12
    Third step: s**12/r**12
    Fourth step: r**6
    Fifth step: s**6
    Sixth step: s**6/r**6
    Seventh step: s**12/r**12-s**6/r**6
    Eighth step: 4*s
    Ninth step: 4*s*e
    Tenth step: 4*s*e*(s**12/r**12-s**6/r**6)

    Total steps: 10
    Performance: O(1) - constant time for constant input

efficient:
    code:
        r2 = r**2
        s2 = s**2
        z = s2/r2
        u = z**3
        return -4*e*u*(1-u)

    First step: r2 = r**2
    Second step: s2 = s**2
    Third step: z = s2/r2
    Fourth step: u = z**3
    Fifth step: 1-u
    Sixth step: -4*e
    Seventh step: (-4*e)*u
    Eighth step: ((-4*e)*u)*(1-u)

    Total steps: 8
    Performance: O(1) - constant time for constant input
```

Conclusion

The efficient algorithm should be around 20% faster than the naive implementation, but both of them run in constant time.

Question 1d

In [3]:
```
import cProfile
print("Naive implementation")
cProfile.run('for i in range(0,1000000): naive_lennard_jones(2.5)') #nai
ve
print("Efficient implementation")
cProfile.run('for i in range(0,1000000): efficient_lennard_jones(2.5,1,
1)') #efficient
```

```
Naive implementation
         1000003 function calls in 1.042 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1000000    0.840    0.000    0.840    0.000 <ipython-input-1-712c776f
7e49>:1(naive_lennard_jones)
        1    0.202    0.202    1.042    1.042 <string>:1(<module>)
        1    0.000    0.000    1.042    1.042 {built-in method builtin
s.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_ls
prof.Profiler' objects}


Efficient implementation
         1000003 function calls in 0.782 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1000000    0.580    0.000    0.580    0.000 <ipython-input-2-d30ff7dd
ce2a>:1(efficient_lennard_jones)
        1    0.202    0.202    0.782    0.782 <string>:1(<module>)
        1    0.000    0.000    0.782    0.782 {built-in method builtin
s.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_ls
prof.Profiler' objects}
```

Conclusion

The efficient implementation is more efficient timewise. The efficient implementation has fewer operations because it only calculates the common factors for the fractions inside the parentheses (saving some work). This is reflected in the cProfile timings for 1M loop iterations.

Question 2

Question 2a

In [4]:
```
import numpy as np
import re

def read_xyz(input_filename):
    '''read_xyz(input_filename)
    Reads an XYZ file to a numpy float32 array.

    Inputs:
        input_filename
            Filename of XYZ file on disk.

            File format:
                line 1: <number of atoms / lines>
                line2: <comment>
                line 3+: <atom type> <x> <y> <z>

    Returns:
        output
            Array of 3D positions of atoms in file.
    '''

    number_of_atoms, comment, output = None, None, []
    with open(input_filename,"r") as fp:
        for line in fp:
            line = line.strip() #clean input
            if number_of_atoms == None:
                number_of_atoms = int(line)
            elif comment == None:
                comment = line
            else:
                splitline = [float(a) for a in re.split(" +",line)[1:]]
                output.append(splitline)
    output = np.array(output,dtype="float32")
    return output
```

Question 2b

In [5]:
```
with open("lj-0003.xyz","w") as fp:
    fp.write('''3
Cambridge Cluster Database
LJ 0.5391356726 0.1106588251 -0.4635601962
LJ -0.5185079933 0.4850176090 0.0537084789
LJ 0.0793723207 -0.4956764341 0.5098517173''')

test_3 = read_xyz("lj-0003.xyz")
test_200 = read_xyz("lj-0200.xyz")
test_500 = read_xyz("lj-0500.xyz")
test_1000 = read_xyz("lj-1000.xyz")
print(test_3)
```

```
[[ 0.53913569  0.11065882 -0.46356019]
 [-0.51850802  0.4850176   0.05370848]
 [ 0.07937232 -0.49567643  0.50985169]]
```

Question 3

Question 3a

In [6]:
```
def number_of_pairwise_interactions(n):
    return int((n**2-n)/2)

print("For 3 atoms:", number_of_pairwise_interactions(3))
print("For 200 atoms:", number_of_pairwise_interactions(200))
print("For 500 atoms:", number_of_pairwise_interactions(500))
print("For 1000 atoms:", number_of_pairwise_interactions(1000))
```

```
For 3 atoms: 3
For 200 atoms: 19900
For 500 atoms: 124750
For 1000 atoms: 499500
```

Question 3b

In [7]:
```
def calculate_pairwise_distance(i,j):
    '''calculate_pairwise_distance(i,j)
    Calculates distance between two sets of 3D coordinates.

    Inputs:
        i - list-like with 3 elements, all floats
            x,y,z position for set 1
        j - list-like with 3 elements, all floats
            x,y,z position for set 2

    Returns:
        distance
            Distance between i and j.
    '''
    return np.sqrt( (j[0]-i[0])**2 + (j[1]-i[1])**2 + (j[2]-i[2])**2 )

def calculate_system_energy(input_array):
    '''calculate_system_energy(input_array)
    Calculates the total potential energy of the system based on atom 3D
 coordinates,
    using the LJ equation (efficient algorithm).

    Inputs:
        input_array
            A numpy.array() of dtype float32, (n rows by 3 columns),
            representing 3D coordiantes of atoms.

    Returns:
        total_energy (float)
            The total energy of the system.
    '''
    total_energy = 0
    num_rows = len(input_array)
    for row1_ix in range(0,num_rows):
        row1 = input_array[row1_ix,:]
        for row2_ix in range(row1_ix+1,num_rows):
            total_energy += efficient_lennard_jones(calculate_pairwise_d
istance(row1,input_array[row2_ix,:]),1,1)
    return total_energy

## Output potential energy calculations
print(calculate_system_energy(test_3)) ## -2.42451546927
print(calculate_system_energy(test_200)) ## -799.624784069
print(calculate_system_energy(test_500)) ## -3315.98663275
print(calculate_system_energy(test_1000)) ## -7017.92348826
```

```
-2.42451546927
-799.624784069
-3315.98663275
-7017.92348826
```

Question 4

Question 4a

In [8]:
```
def calculate_system_energy_naive_cutoff(input_array, cutoff_dist=1):
    '''calculate_system_energy_naive_cutoff(input_array)
    Calculates the total potential energy of the system based on atom 3D
 coordinates,
    using the LJ equation (efficient algorithm).

    Inputs:
        input_array
            A numpy.array() of dtype float32, (n rows by 3 columns),
            representing 3D coordiantes of atoms.
        cutoff_dist
            Distance within we will consider pairwise interactions. Beyo
nd this,
            the energy will not be calculated.

    Returns:
        total_energy (float)
            The total energy of the system.
    '''
    total_energy = 0
    num_rows = len(input_array)
    for row1_ix in range(0,num_rows):
        row1 = input_array[row1_ix,:]
        for row2_ix in range(row1_ix+1,num_rows):
            r = calculate_pairwise_distance(row1,input_array[row2_ix,:])
            if r <= cutoff_dist:
                total_energy += efficient_lennard_jones(r,1,1)
    return total_energy

print(calculate_system_energy_naive_cutoff(test_3)) ## 0
print(calculate_system_energy_naive_cutoff(test_200)) ## 37.8305661135
print(calculate_system_energy_naive_cutoff(test_500)) ## 30.5643572661
print(calculate_system_energy_naive_cutoff(test_1000)) ## 58.5072399514
```

```
0
37.8305661135
30.5643572661
58.5072399514
```

Conclusion

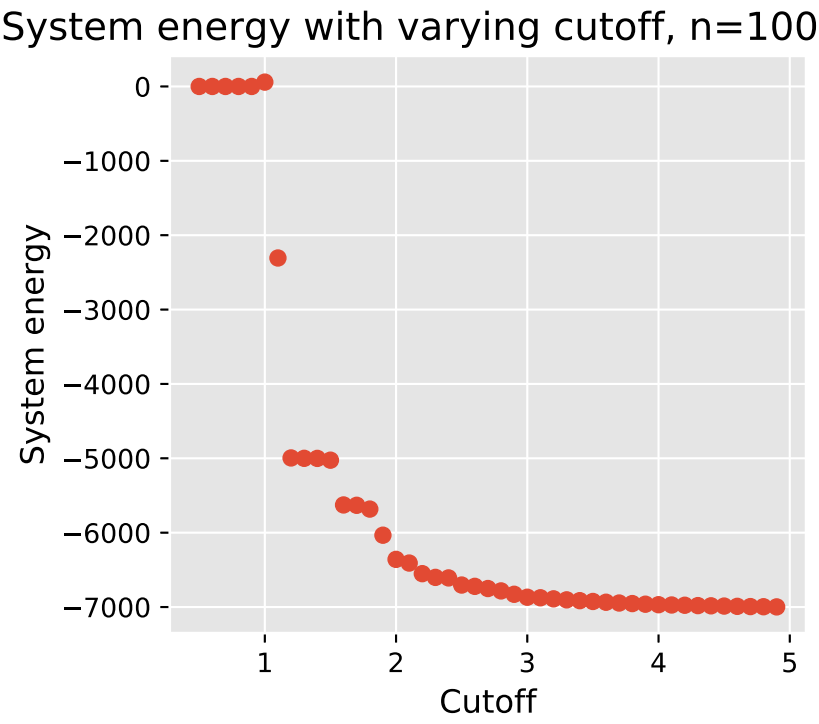
No, this is a terrible cutoff. Any values with distances below 1 have a positive energy, and above 1 have a negative energy. The values are way off from the real values.

Question 4b

In [9]:
```
cutoff_results = []

for cutoff in range(5,50):
    cutoff = cutoff/10
    res = (cutoff,calculate_system_energy_naive_cutoff(test_1000,cutoff
))
    print(res)
    cutoff_results.append(res)

cutoff_results = np.array(cutoff_results)
```

```
(0.5, 0)
(0.6, 0)
(0.7, 0)
(0.8, 0)
(0.9, 0)
(1.0, 58.50723995135499)
(1.1, -2306.512680215119)
(1.2, -4995.1024525203129)
(1.3, -5000.1721882622924)
(1.4, -5001.1699938772845)
(1.5, -5025.0920934338328)
(1.6, -5627.1969826076784)
(1.7, -5631.825118612529)
(1.8, -5682.7524489700882)
(1.9, -6033.937774074423)
(2.0, -6358.973914454401)
(2.1, -6408.4642615616322)
(2.2, -6551.1959376911591)
(2.3, -6601.067007052111)
(2.4, -6608.3495103767373)
(2.5, -6704.1623075799225)
(2.6, -6721.7050311238127)
(2.7, -6751.010877919568)
(2.8, -6783.1506269241863)
(2.9, -6828.2563850977431)
(3.0, -6867.9354360142379)
(3.1, -6875.4644610084233)
(3.2, -6889.7332767907765)
(3.3, -6903.3188015144442)
(3.4, -6914.3017886532571)
(3.5, -6925.0270770113257)
(3.6, -6935.3613955591682)
(3.7, -6945.8938799311036)
(3.8, -6953.3731458138163)
(3.9, -6961.7257349350384)
(4.0, -6968.6521231016659)
(4.1, -6972.3751785731065)
(4.2, -6975.8133441936661)
(4.3, -6980.6769075559932)
(4.4, -6983.6046586565599)
(4.5, -6986.854141150764)
(4.6, -6990.5663203287859)
(4.7, -6993.8461886636387)
(4.8, -6995.8489052128843)
(4.9, -6997.9978726585614)
```

In [19]:
```
import matplotlib.pyplot as plt
from matplotlib import rcParams
from IPython.display import set_matplotlib_formats
%matplotlib inline
set_matplotlib_formats('svg')

plt.style.use('ggplot')
rcParams['figure.figsize'] = (5,4)
rcParams['figure.dpi'] = 200
rcParams['font.family'] = 'DejaVu Sans'

COLOR = 'black'
rcParams['text.color'] = COLOR
rcParams['axes.labelcolor'] = COLOR
rcParams['xtick.color'] = COLOR
rcParams['ytick.color'] = COLOR

plt.scatter(cutoff_results[:,0],cutoff_results[:,1])
plt.title("System energy with varying cutoff, n=1000 atoms")
plt.xlabel("Cutoff")
plt.ylabel("System energy")
plt.tight_layout()
plt.savefig("problem_4b.pdf")
plt.show()
```



Conclusion

By the chart and the results, the point on the graph where it is at 99% of the real system energy is at -7017*0.99 = -6,947 or around a cutoff of 3.7.

Question 4c

In [11]:
```
ideal_cutoff = 3.7

cProfile.run('calculate_system_energy(test_3)') #naive
cProfile.run('calculate_system_energy_naive_cutoff(test_3,ideal_cutoff)'
) #naive cutoff

cProfile.run('calculate_system_energy(test_200)') #naive
cProfile.run('calculate_system_energy_naive_cutoff(test_200,ideal_cutof
f)') #naive cutoff

cProfile.run('calculate_system_energy(test_500)') #naive
cProfile.run('calculate_system_energy_naive_cutoff(test_500,ideal_cutof
f)') #naive cutoff

cProfile.run('calculate_system_energy(test_1000)') #naive
cProfile.run('calculate_system_energy_naive_cutoff(test_1000,ideal_cutof
f)') #naive cutoff
```

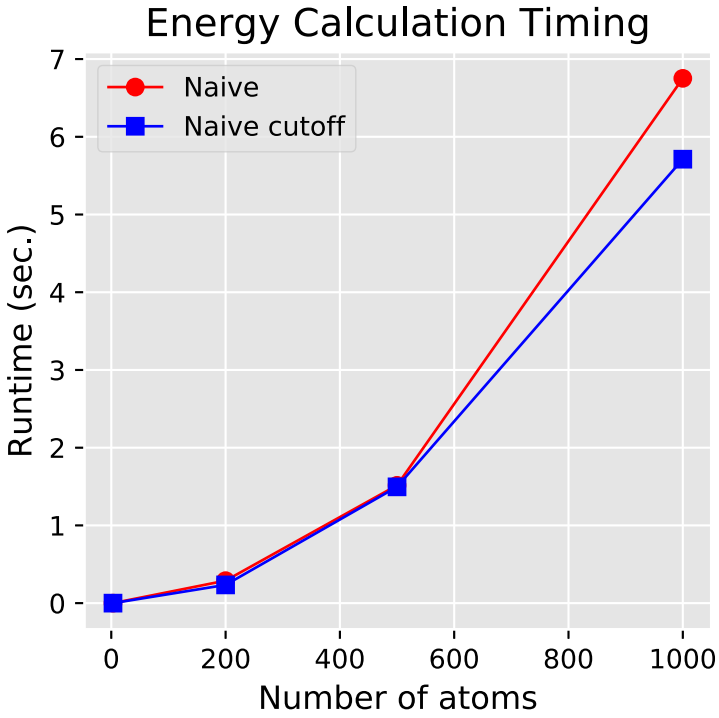
```
         11 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        3    0.000    0.000    0.000    0.000 <ipython-input-2-d30ff7dd
ce2a>:1(efficient_lennard_jones)
        3    0.000    0.000    0.000    0.000 <ipython-input-7-f02a3be6
0cca>:1(calculate_pairwise_distance)
        1    0.000    0.000    0.000    0.000 <ipython-input-7-f02a3be6
0cca>:17(calculate_system_energy)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {built-in method builtin
s.exec}
        1    0.000    0.000    0.000    0.000 {built-in method builtin
s.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_ls
prof.Profiler' objects}


         11 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        3    0.000    0.000    0.000    0.000 <ipython-input-2-d30ff7dd
ce2a>:1(efficient_lennard_jones)
        3    0.000    0.000    0.000    0.000 <ipython-input-7-f02a3be6
0cca>:1(calculate_pairwise_distance)
        1    0.000    0.000    0.000    0.000 <ipython-input-8-aa37609c
8e31>:1(calculate_system_energy_naive_cutoff)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {built-in method builtin
s.exec}
        1    0.000    0.000    0.000    0.000 {built-in method builtin
s.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_ls
prof.Profiler' objects}


         39805 function calls in 0.271 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    19900    0.033    0.000    0.033    0.000 <ipython-input-2-d30ff7dd
ce2a>:1(efficient_lennard_jones)
    19900    0.218    0.000    0.218    0.000 <ipython-input-7-f02a3be6
0cca>:1(calculate_pairwise_distance)
        1    0.020    0.020    0.271    0.271 <ipython-input-7-f02a3be6
0cca>:17(calculate_system_energy)
        1    0.000    0.000    0.271    0.271 <string>:1(<module>)
        1    0.000    0.000    0.271    0.271 {built-in method builtin
s.exec}
        1    0.000    0.000    0.000    0.000 {built-in method builtin
s.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_ls
```

```
prof.Profiler' objects}


         28172 function calls in 0.231 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     8267    0.013    0.000    0.013    0.000 <ipython-input-2-d30ff7dd
ce2a>:1(efficient_lennard_jones)
    19900    0.202    0.000    0.202    0.000 <ipython-input-7-f02a3be6
0cca>:1(calculate_pairwise_distance)
        1    0.016    0.016    0.231    0.231 <ipython-input-8-aa37609c
8e31>:1(calculate_system_energy_naive_cutoff)
        1    0.000    0.000    0.231    0.231 <string>:1(<module>)
        1    0.000    0.000    0.231    0.231 {built-in method builtin
s.exec}
        1    0.000    0.000    0.000    0.000 {built-in method builtin
s.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_ls
prof.Profiler' objects}


         249505 function calls in 1.570 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   124750    0.195    0.000    0.195    0.000 <ipython-input-2-d30ff7dd
ce2a>:1(efficient_lennard_jones)
   124750    1.267    0.000    1.267    0.000 <ipython-input-7-f02a3be6
0cca>:1(calculate_pairwise_distance)
        1    0.108    0.108    1.570    1.570 <ipython-input-7-f02a3be6
0cca>:17(calculate_system_energy)
        1    0.000    0.000    1.570    1.570 <string>:1(<module>)
        1    0.000    0.000    1.570    1.570 {built-in method builtin
s.exec}
        1    0.000    0.000    0.000    0.000 {built-in method builtin
s.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_ls
prof.Profiler' objects}


         158141 function calls in 1.377 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    33386    0.052    0.000    0.052    0.000 <ipython-input-2-d30ff7dd
ce2a>:1(efficient_lennard_jones)
   124750    1.235    0.000    1.235    0.000 <ipython-input-7-f02a3be6
0cca>:1(calculate_pairwise_distance)
        1    0.090    0.090    1.377    1.377 <ipython-input-8-aa37609c
8e31>:1(calculate_system_energy_naive_cutoff)
        1    0.000    0.000    1.377    1.377 <string>:1(<module>)
        1    0.000    0.000    1.377    1.377 {built-in method builtin
s.exec}
```

```
        1    0.000    0.000    0.000    0.000 {built-in method builtin
s.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_ls
prof.Profiler' objects}


         999005 function calls in 6.615 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   499500    0.817    0.000    0.817    0.000 <ipython-input-2-d30ff7dd
ce2a>:1(efficient_lennard_jones)
   499500    5.330    0.000    5.330    0.000 <ipython-input-7-f02a3be6
0cca>:1(calculate_pairwise_distance)
        1    0.468    0.468    6.615    6.615 <ipython-input-7-f02a3be6
0cca>:17(calculate_system_energy)
        1    0.000    0.000    6.615    6.615 <string>:1(<module>)
        1    0.000    0.000    6.615    6.615 {built-in method builtin
s.exec}
        1    0.000    0.000    0.000    0.000 {built-in method builtin
s.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_ls
prof.Profiler' objects}


         575642 function calls in 5.770 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    76137    0.128    0.000    0.128    0.000 <ipython-input-2-d30ff7dd
ce2a>:1(efficient_lennard_jones)
   499500    5.266    0.000    5.266    0.000 <ipython-input-7-f02a3be6
0cca>:1(calculate_pairwise_distance)
        1    0.376    0.376    5.770    5.770 <ipython-input-8-aa37609c
8e31>:1(calculate_system_energy_naive_cutoff)
        1    0.000    0.000    5.770    5.770 <string>:1(<module>)
        1    0.000    0.000    5.770    5.770 {built-in method builtin
s.exec}
        1    0.000    0.000    0.000    0.000 {built-in method builtin
s.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_ls
prof.Profiler' objects}
```

In [20]:

```
import time

def runtime(function):
    t0 = time.time()  # start time
    exec(function)
    t1 = time.time() # end time
    return(t1-t0)

n3 = runtime('calculate_system_energy(test_3)') #naive
c3 = runtime('calculate_system_energy_naive_cutoff(test_3,ideal_cutoff)'
) #naive cutoff

n200 = runtime('calculate_system_energy(test_200)') #naive
c200 = runtime('calculate_system_energy_naive_cutoff(test_200,ideal_cuto
ff)') #naive cutoff

n500 = runtime('calculate_system_energy(test_500)') #naive
c500 = runtime('calculate_system_energy_naive_cutoff(test_500,ideal_cuto
ff)') #naive cutoff

n1000 = runtime('calculate_system_energy(test_1000)') #naive
c1000 = runtime('calculate_system_energy_naive_cutoff(test_1000,ideal_cu
toff)') #naive cutoff

rcParams['figure.figsize'] = (4,4)

plt.plot([3,200,500,1000],[n3,n200,n500,n1000],"ro-",linewidth="1")
plt.plot([3,200,500,1000],[c3,c200,c500,c1000],"bs-",linewidth="1")
plt.title("Energy Calculation Timing")
plt.xlabel("Number of atoms")
plt.ylabel("Runtime (sec.)")
plt.legend(["Naive","Naive cutoff"])
plt.tight_layout()
plt.savefig("problem_4c.pdf")
plt.show()
```



Conclusion

No, we have not solved the N2 scaling problem, since we are still calculating all pairwise distance between atom pairs (and therefore need to do N2 array accesses). We have only slightly reduced the runtime of the energy calculation. Idea: Perhaps we could reduce the complexity by calculating distances to a known point (e.g. the origin), sorting the values, and then only calculating the nearest pairs within a sliding window.

Question 5

Give the tilde approximations for the following quantities.

a) N + 1

Answer: N

b) 1 + 1/N

Answer: 1

c) (1 + 1/N)(1 + 2/N)

Answer: 1

d) 2N^3 - 15N^2 + N

Answer: 2N^3

e) log(2N)/log(N)

Answer: 1 (since top goes to log(2)+log(N))

f) log(N^2+1)/log(N)

Answer: 2 (since for large N top goes to 2*log(N))

g) N^100/(2^N)

Answer: 1 (FOR N>>100, 2^N rises faster than N^100, function goes to 0)

Question 6

Give the order of growth (as a function of *N*) of the running times of each of the following code fragments.

Question 6a

```
def problem_a(n):
    sum = 0
    k = n
    while k > 0: ## N^2 * 2/N = 2N
        for i in range(k): ##N
            sum += 1
        k = k // 2 ## 2/N
    return sum
```

Answer: 2N. The program loops through all N the first round, then loops through the series where k is N/2,N/4... until k is 1. this series adds up to N so the total time is 2*N.

Question 6b

```
def problem_b(n):
    sum = 0
    i = 1
    while i < n: ##N
        for j in range(i): ##N
            sum += 1
        i = i * 2
    return sum
```

Answer: N. This is the same fractional series up to N from 1 as in problem a, which adds up to N. The program doesn't run through all N at the beginning since i is 1. Also, because i is discrete powers of 2 under N, it runs only in # of iterations of powers of 2 so it is only ~N.

Question 6c

```
def problem_c(n):
    sum = 0
    i = 1
    while i < n: ##log(N)
        for j in range(n): ##N*log(N)
            sum += 1
        i = i * 2
    return sum
```

Answer: N*log(N). The inner loop runs from 0 through N many times, and the number of times it does so is log_2(N) since the number of loop iterations in the outer loop is determined by i < n where i can only be 2^(1,2,3...) up to n, therefore this number of outer loop iterations is 2^x = N or log2(N) = x.

Question 7

Give a formula to predict the running time of a program for a problem of size *N* when doubling experiments have shown that the doubling factor is 2^b and the running time for problems of size N_0 is *T*.

$$f(N_0) = T_0$$
$$f(2 * N_0) = 2^b * T_0$$
$$f(2^2 * N_0) = 2^{2b} * T_0$$
$$\therefore Answer : f(N) = 2^{b(\frac{N}{N_0} - 1)} * T_0$$

Question 8

Implement BetterChange (from Pevzner, pp. 21). This is the USChange problem but for any denomination. Note that *M* is a float, **c** is a vector of denomination values, and *d* is a positive integer corresponding to the number of denominations.

BETTERCHANGE(*M*, **c**, *d*)
r ← M
for k←1 to d:
  $i_k \leftarrow$ r/c_k
  r ← (r - c_k * i_k)
return (i_1, i_2, ..., i_d)

a) Implement the above method in python.
b) Compute BetterChange() with M = 0.79, c = [1, 3, 9], and d = 3.
c) What is the runtime of BetterChange as a function of M, c, and d?
d) Provide a case (specify a value of M, c, and d) where BetterChange is correct.
e) Provide a case (specify a value of M, c, and d) where BetterChange is incorrect.

Question 8a

In [13]:
```
def betterchange(M,c,d):
    r = M ##amount of money remaining after change made
    i = [] ##array to store output values
    ## ASSUME: c array is decreasing in values
    for k in range(0,d):
        i.append(r // c[k])
        r = r - c[k] * i[k]
    return i
```

Question 8b

In [14]:
```
betterchange(0.79,[.09,.03,.01],3) ##modified input params after discuss
ion with Kevin Bu
```

Out[14]:
```
[8.0, 2.0, 1.0]
```

Question 8c

The runtime of betterchange is proportional to d since the algorithm loops and only always loops through all elements in c which is length d.

Question 8d

In [15]:
```
## Correct version of 8b
betterchange(0.79,[.09,.03,.01],3)
```

Out[15]:
```
[8.0, 2.0, 1.0]
```

Question 8e

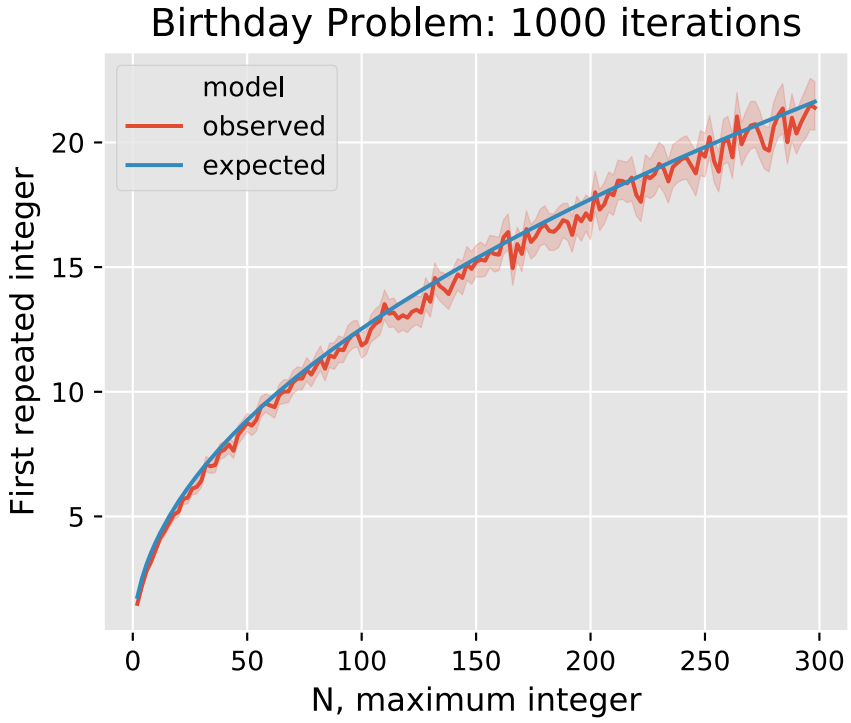
In [16]:
```
## incorrect - not the right amount of change (there is a remainder)
## and because c is out of order, there are more coins than necessary
betterchange(0.67,[.05,.10,.25],3)
```

Out[16]:
```
[13.0, 0.0, 0.0]
```

Question 9

Birthday problem

In [21]:
```
import matplotlib.pyplot as plt
from matplotlib import rcParams
%matplotlib inline
import numpy as np
import seaborn as sb
import pandas as pd

plt.style.use('ggplot')
rcParams['figure.figsize'] = (5,4)
rcParams['figure.dpi'] = 200
rcParams['font.family'] = 'DejaVu Sans'

COLOR = 'black'
rcParams['text.color'] = COLOR
rcParams['axes.labelcolor'] = COLOR
rcParams['xtick.color'] = COLOR
rcParams['ytick.color'] = COLOR

def generate_random_integers(N,length):
    return np.random.randint(low=0,high=N,size=length)

def test_birthday_problem(N,rounds=1000):
    lengths = []
    for round in range(0,rounds):
        test_set = set()
        while True:
            num = np.random.randint(low=0,high=N)
            if num not in test_set:
                test_set.add(num)
            else:
                lengths.append(len(test_set))
                break
    return lengths

results = []
for i in range(2,300,2):
    #results: length iteration type first_repeat
    for a,v in enumerate(test_birthday_problem(i,500)):
        results.append([i,a,"observed",v])
        results.append([i,a,"expected",np.sqrt(np.pi*i/2)])

df = pd.DataFrame(results,columns=["length","iteration","model","first_r
epeat"])
sb.lineplot(data=df,x="length",y="first_repeat", hue="model")
plt.title("Birthday Problem: 1000 iterations")
plt.xlabel("N, maximum integer")
plt.ylabel("First repeated integer")
plt.savefig("problem_9.pdf")
plt.show()
```



Conclusion

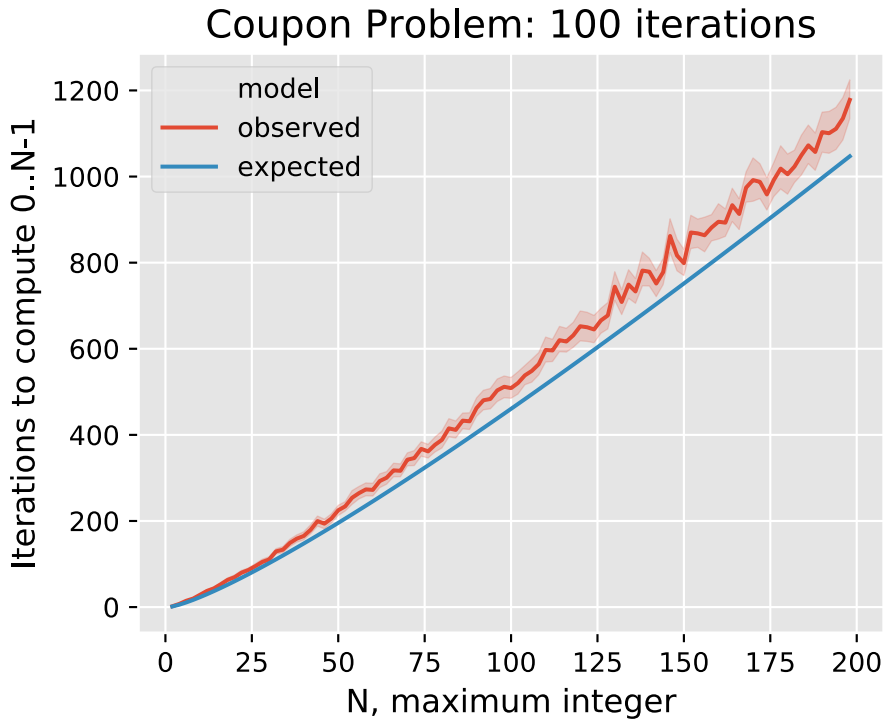
As shown by the graph above of 1000 iterations of the birthday problem on sequences with max integer N from 2 to 200, the model ~ $\sqrt{\pi * \frac{N}{2}}$ accurately represents the observed counts of the positions of the first repeats in the sequences of random integers (it stays within the confidence interval even out to 200 iterations).

Question 10

Coupon collector problem

In [23]:
```
def test_coupon_problem(N,rounds=1000):
    lengths = []
    for round in range(0,rounds):
        len_sequence = 0
        test_set = set()
        while True:
            num = np.random.randint(low=0,high=N)
            test_set.add(num)
            if len(test_set) != N:
                len_sequence += 1
            else:
                lengths.append(len_sequence)
                break
    return lengths
```

In [24]:
```
results = []
for i in range(2,200,2):
    #results: length iteration type first_repeat
    for a,v in enumerate(test_coupon_problem(i,100)):
        results.append([i,a,"observed",v])
        results.append([i,a,"expected",i*np.log(i)])

df = pd.DataFrame(results,columns=["length","iteration","model","first_r
epeat"])
sb.lineplot(data=df,x="length",y="first_repeat", hue="model")
plt.title("Coupon Problem: 100 iterations")
plt.xlabel("N, maximum integer")
plt.ylabel("Iterations to compute 0..N-1")
plt.savefig("problem_10.pdf")
plt.show()
```



Conclusion

As shown in the graph, the observed growth of the function to compute a set of numbers from 0 to N-1 by selecting random numbers seems to be close to the function ~NH_N (while it may slightly underestimate it, it is within a very small number). Therefore, the model is validated.