BMI 2005 Homework #2
Due February 13, 2019

Homework must be submitted to Blackboard using a Jupyter notebook!

1) As we discussed in class, simulation of a "real" liquid, such as a box of non-bonded particles at a given temperature and pressure, requires consideration of so-called periodic boundary conditions. In our simulation, we will assume the center of the box is at position 0,0,0 and that the initial vectors describing the extent of the box are equal (e.g., this box is a square!).

   Attached to this homework is a new XYZ file defining a box of 200 Lennard-Jones particles. The boundary vector for this box for a reasonable box temperature and pressure (we'll get there later, don't worry about this yet…) is 3.21829795. The total length of the box is twice the vector length in each dimension, so this box is 6.4365959 in length in each dimension (x, y, and z).

   a) Implement a new total LJ potential function (as you did for problem 4(a) in the previous homework) which includes consideration of a periodic boundary, and compute the total potential for the system in the supplied new XYZ file (lj-0200-liquid.xyz).

      This will require you to correct the distance between atoms in each dimension. You may in the previous homework have written something like the following, which computes the distance based on the vector norm between the x, y, and z positions of two atoms, and the conditional defining your cutoff distance:

```
def distance(a, b):
  return np.linalg.norm(a - b)

def V_ij(r):
  return 4 *((1/r)**12 - (1/r)**6)

def lj_cut(atoms, r_cut):
  n_atoms = a.shape[0]
  LJ_potential = 0.0
  for i in range(n_atoms):
    for j in range(i + 1, n_atoms):
      r = distance(a[i], a[j])
      if r < r_cut:
        LJ_potential += V_ij(r)
  return LJ_potential
```

Because you will need to correct the distance between atoms in each dimension, based on the periodicity of the system. So this approach will not quite work: you will need to consider the distance in each dimension between the two particles in your double loop. In this case, it is easiest to consider the square euclidian distance, because it allows you to easily accumulate the distance in each dimension and then simply take a square root at the end. E.g.,

$$d^2(\mathbf{p}, \mathbf{q}) = (p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2 + \cdots + (p_n - q_n)^2.$$

In pseudowords, a good approach to this is the following:

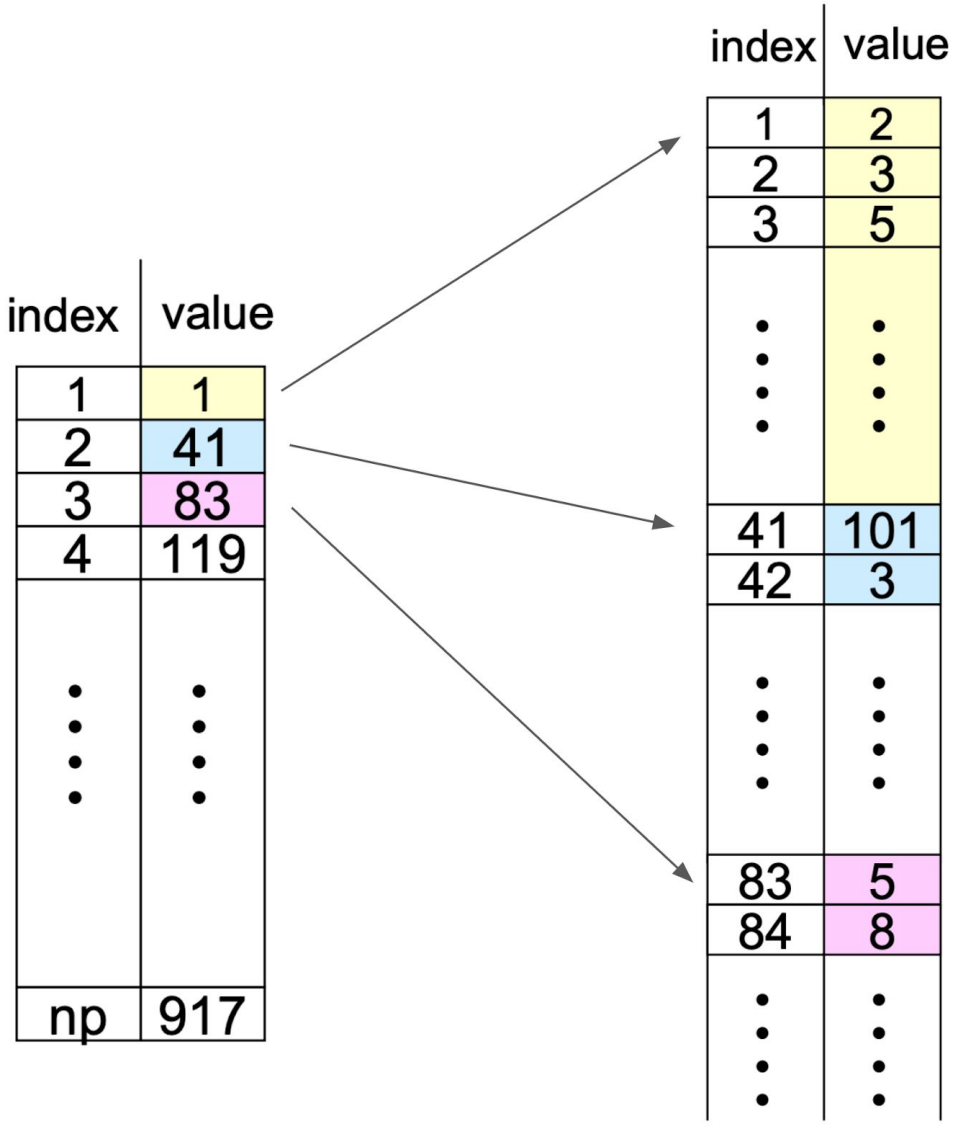
```
for each atom i in the system
  for each pairwise interaction atom j
    square_distance ← 0.0
    for each dim
      distance_in_dim ← a[i][dim] - a[j][dim]
      distance_in_dim -= boundary_length[dim] * \
        round(distance_in_dim / boundary_length[dim])
      square_distance += distance_in_dim ** 2
    distance = sqrt(square_distance)
    if (distance < cutoff...)...
```

Where `distance_in_dim` is modified using the supplied function.

b) Compute the new potential energy for the system using your new function and a cutoff value of 2.0. Report the value.

2) We discussed two methods of list construction, the Verlet list, and the cell list, which can be implemented by an elementary hash table, and a linked list, respectively.

a) First, we will implement a static Verlet list and rewrite our potential energy function once again to accommodate this new data structure. Constructing the list involves looping over all atom pairs, determining which pairs are within the cutoff + sheath distance, and constructing two arrays: a lookup table, where the location of pair information can be found for each atom, and a value table, where all pair atoms are found. The number of pairwise interactions per atom can be determined purely by the values in the lookup table, by comparing the position in the value table for the current and next atom.



Just like our previous attempt at a new potential function, periodicity must be considered. A pseudoword attempt at the list function might look like, for a cutoff of 2.0 and a sheath of 0.3:

```
cutoff ← 2.0
sheath ← 0.3
cutoff_plus_sheath ← cutoff + sheath
lookup ← empty numpy array of integers, length n_atoms
value ← unknown length, placeholder array
pair_counter ← 0
for each atom i in the system
  for each pairwise interaction atom j
    square_distance ← 0.0
```

```
    for each dim
      distance_in_dim ← a[i][dim] - a[j][dim]
      distance_in_dim -= boundary_length[dim] * \
        round(distance / boundary_length[dim])
      square_distance += distance_in_dim ** 2
    distance = sqrt(square_distance)
    if (distance < cutoff_plus_sheath)
      pair_counter ← pair_counter + 1
      value ← push pair index j onto array
  lookup[i] ← pair_counter
```

b) After implementing the above pairwise function to generate a Verlet list, implement a total potential energy function that takes as input both arrays in the Verlet list, with appropriate control logic to only include those atoms within the cutoff but not within the sheath. Argue for why the complexity of this function is now O(N).

3) Now we will tackle the linked cell formulation, which is the first method we will cover to transform this problem into a true linear or logarithmetic problem. We will start with a very simple implementation of the cell list method, simply creating an appropriate set of linked lists, and continue to improve on our implementation with time once we start actually moving atoms in the simulation.

   a) Given a fixed box length of 6.4365959 in all directions, and a cutoff of 2.0, what would be an appropriate size for a cell-based domain decomposition of this box? Why? (Hint: we covered this in the 4th week of class).

   b) Create an array of linked lists for a 3x3x3 (27 lists) decomposition of the array. How you index the position of each cell is up to you, but you need to know which area of the box each cell represents! Each node in each of the linked list should contain at a minimum a index value (the index of the atom) and a pointer to next.

   c) Then,

      i) Create an array or other data structure that specifies which cell neighbors are adjacent. Remenber, in 3D each cell is 26 neighbors with periodic boundary conditions!
      ii) Implement an append function which allows you to easily add a new node to an arbitrary list by supplying the appropriate index to instantiate the node within the appropriate cell list. This method should handle the case where the linked list is empty and the head of the list has not yet been assigned.

iii) Loop over all atoms in the simulation and assign them to a cell by instantiating a node in each cell as appropriate.

4) In class, we discussed the extension of the 1D peak problem to 2 dimensions. Write an implementation of two_peak() that returns a peak (as defined by an entry greater than or equal to its 4 neighbor values). Run it on the given array, and assess the complexity in terms of O(n) (in theory, not with doubling experiments unless you'd prefer!).

```
import numpy as np
A = np.array([
    [1, 3, 10, 2],
    [14, 13, 12, 5],
    [15, 9, 11, 6],
    [16, 17, 13, 18]
])
```

5) Conduct doubling experiments to compare the run time of selection vs insertion vs merge vs quick (randomized vs non-randomized) in various cases. Plot the results (all 5 sorts per case of array) for array sizes n = 2, 4, ... 1024. The functions from lab are provided below - helper 'wrapper' functions for quicksort (randomized and nonrandomized) are also included so that all the sort functions are a function of an input array A only. The five cases are as follows:
    a) A is perfectly sorted and contains random integers (0 to 100 inclusive).
    b) A is reversely sorted and contains random integers (0 to 100 inclusive)
    c) A has a random order of integers from 0 to 100 inclusive.
    d) A is made of entirely identical entries (all 1s).
    e) A is composed of 3 distinct entries (in random order) i.e. 0, 1 and 2.

    Do the results agree with what you expect?

```
import time
import numpy as np

def selection_sort(A):
    for i in range(len(A)):
        min_idx = i
        for j in range(i + 1, len(A)):
            if A[min_idx] > A[j]:
                min_idx = j
        A[i], A[min_idx] = A[min_idx], A[i]
    return A

def insertion_sort(A):
    n = len(A)
```

```
    count = 0
    for i in range(1,n):
        k = i
        while k > 0 and A[k] < A[k-1]:
            A[k], A[k-1] = A[k-1], A[k]
            k = k - 1
            count += 1
    return count, A

def merge (front, back):
    pos_f, pos_b = 0,0
    merged = np.zeros(len(front)+len(back))
    for i in range (len(merged)):
        if pos_f == len(front):
            merged[i] = back[pos_b]
            pos_b += 1
        elif pos_b == len(back):
            merged[i] = front[pos_f]
            pos_f += 1
        elif front[pos_f] < back[pos_b]:
            merged[i] = front[pos_f]
            pos_f += 1
        else:
            merged[i] = back[pos_b]
            pos_b += 1
    return merged

def merge_sort(A):
    n = len(A)
    if n <= 1:
        return A
    mid = int(n/2)
    front = merge_sort(A[0:mid])
    back = merge_sort(A[mid:])
    return merge(front, back)

def partition(A, lo, hi):
    pivotvalue = A[lo]
    i = lo+1
    j = hi
    done = False
    while not done:
        while i <= j and A[i] <= pivotvalue:
```

```
            i = i + 1
        while A[j] >= pivotvalue and j >= i:
            j = j - 1
        if j < i:
            done = True
        else:
            A[i], A[j] = A[j], A[i]
    A[lo], A[j] = A[j], A[lo]
    return j, A

def quick_sort(A, lo, hi):
    if lo < hi:
        splitpoint, A = partition(A, lo, hi)
        A = quick_sort(A, lo, splitpoint - 1)
        A = quick_sort(A, splitpoint + 1, hi)
    return A

def quick_sort_helper_rand(A):
    np.random.shuffle(A)
    return quick_sort(A, 0, len(A)-1)

def quick_sort_helper_nonrand(A):
    return quick_sort(A, 0, len(A)-1)
```

6) Solve the following recurrences (i.e. find a closed form expression for n).

a) $S(0) = 6$ for $n = 0$
$S(n) = S(n-1) + 2$ for $n = 1, 2, 3, ...$

b) $T(1) = 2$ for $n = 1$
$T(n) = 2T(n-1) + 4$ for $n = 2, 3, 4, \ldots$

c) **BONUS**
$Q(1) = c$ for $n = 1$
$Q(n) = Q(n/2) + 2n$ for $n = 2, 4, 8, \ldots$

7) The number of comparisons in merge sort can be written as

$T(1) = 0$ for $n = 1$,
(a subarray of 1 element requires no comparisons)

$T(n) = 2T(n/2) + (n-1)$ for $n = 2, 4, 8 \ldots$
(derived from $T(n/2)$ comparisons to get the first half of the array, another $T(n/2)$ for the second half and $n-1$ to merge them)

What is a closed form expression for T(n)?

8) Here we will work towards a better quicksort algorithm based on modifications discussed in the lab.

   a) Draw traces to show what happens when you pivot on using 13 as the pivot when partition (as defined in the lab) is called i.e.

```
A = [13, 19, 9, 5, 9, 8, 11, 9, 6]
partition(A, lo = 0, hi = len(A)-1)
```

   b) So far we've discussed quicksort that partitions each array into two components based on a pivot v (one portion greater than or equal to v, the other half less than or equal to v). However, such a partition can lead to potentially poor performances when the array contains duplicates. Here we implement a 3-way quicksort that partitions the array into thirds. Convert the following from pseudocode to Python.

```
quicksort_3way(A, lo, hi)
    if hi <= lo
        return A
    lt, i, gt = lo, lo + 1, hi
    v = A[lo]
    while i <= gt
        if A[i] < v
            exchange A[lt], A[i]
            increase i and lt by 1
        elif A[i] > v
            exchange A[i], A[gt]
            decrease gt by 1
        else
            increase i by 1
    A <- quicksort_3way(A, lo, lt - 1)
    A <- quicksort_3way(A, gt + 1, hi)
return A
```

   c) What value does `i` terminate on (in the first outermost recursive call) when quicksort_3way runs with pivot value 2 on the following array (using the first entry as a pivot)?

```
A = [2, 3, 1, 2, 2, 2, 3, 3]
quicksort_3way(A, 0, len(A) - 1)
```

9) Consider the following sorting algorithm:

```
Funny-sort (A, i,j):
    if A[i] > A[j]
        Exchange A[i] <-> A[j]
    if (i + 1)>= j
        return
    k <- int((j-i+1)/3) # the "int" function acts as floor
    Funny-sort(A, i, j-k)
    Funny-sort(A, i+k, j)
    Funny-sort(A, i, j-k)
```

a) Does this correctly sort the array? Why?
b) Give a recurrence for this (e.g. something like T(n) = ?)
c) Ignoring constant factors and smaller terms, guess whether the worst case (O(n)) running time is better, the same, or worse than MergeSort?
d) **BONUS:** Can you figure out the worst case run time? (e.g. O(?)) HINT: The master theorem says that given a recurrence of form aT(n/b) + f(n) where a, b are constants greater than or equal to 1 and $f(n) = n^c$, then in the following 3 cases:
   (1) if $c > \log_b(a)$, then $T(n) = O(n^c)$
   (2) if $c = \log_b(a)$, then $T(n) = O(n^c \log(n))$
   (3) if $c < \log_b(a)$, then T(n) = O(n^$\log_b(a)$)