M345SC 2019 Homework 1 Solution

Part 1. (8 pts)

Here, you will work with strings corresponding to DNA sequences. Consider a DNA sequence, *S*, provided as input. The goal is to find: 1) frequently-occurring *k-mers* within *S*, 2) their positions in the input sequence, and 3) the number of point-x mutations of these k-mers that are in *S*. Note that a *k-mer* is simply a length-k string of consecutive nucleotides in *S*, and a point-x mutation of a k-mer, *s*, is another k-mer which is identical to *s* aside from the nucleotide in its *xth* position (with x=0 corresponding to the first nucleotide in a k-mer). So, the sequence "ACGTCG" contains 4 *3-mers* (ACG, CGT, GTC, TCG) and TCG is a point-0 mutation of ACG (and vice-versa).

1) Add code to the function, *ksearch* in *p1_dev.py* so that it finds all *k-mers* that occur at least *f* times within the the input sequence, *S* (and with *f* and *k* also provided as input). The function should return the locations of these frequent k-mers as well as the *k-mer* strings themselves.

2) Complete *ksearch* by computing the total number of k-mers which are point-x mutations of each "frequently-occurring" k-mer found in 1) with *x* provided as an input parameter.

Ultimately, the function should return three lists: L1 containing the strings corresponding to the frequently-occurring k-mers; L2 containing the locations of the frequent k-mers, specifically, the *ith* element in L2 should be a list of integers corresponding to the indices in *S* where the *ith* k-mer in L1 is found; L3 containing the number of point-x mutations for each k-mer in L1. If no frequent k-mers are found, simply return three empty lists.

Ans:

Code for a dictionary-based implementation:

```
def ksearch(S,k,f,x):
    """
    Search for frequently-occurring k-mers within a DNA sequence
    and find the number of point-x mutations for each frequently
    occurring k-mer.
    Input:
    S: A string consisting of A,C,G, and Ts
    k: The size of k-mer to search for (an integer)
    f: frequency parameter -- the search should identify k-mers which
    occur at least f times in S
    x: the location within each frequently-occurring k-mer where
    point-x mutations will differ.

    Output:
    L1: list containing the strings corresponding to the frequently-occurring k-mers
    L2: list containing the locations of the frequent k-mers
    L3: list containing the number of point-x mutations for each k-mer in L1.

    Discussion: Add analysis here
    """

    #Build dictionary of k-mers
    D = {}
    n = len(S)
    for i in range(n-k+1):
        if i%10**6==0: print('i=',i)
        key = S[i:i+k]
        if key in D:
            D[key].append(i)
        else:
            D[key]=[i]
```

```
    #Extract frequently-occurring k-mers
    L1,L2=[],[]
    for l,v in D.items():
        if len(v)>=f:
            L1.append(l)
            L2.append(v)

    #Find point-x mutations
    L3=[]
    for l in L1:
        L3.append(0)
        for c in "ACGT":
            if c != l[x]:
                l2 = l[:x] + c + l[x+1:]
                if l2 in D: L3[-1]+=len(D[l2])

    return L1,L2,L3
```

3) Carefully analyze the running time of your code in *ksearch*. Your analysis should include: a) a concise description of your algorithm and b) a clear and concise discussion of the running time (including the leading-order running time) and how it depends on the input. We are generally interested in cases where the length of S is very large though you may assume that a string containing all k-mers for S will fit in your computer's main memory.

Ans:

The basic approach is to iterate through S and build a hash table that contains a list of indices for each unique k-mer. Then, iterating through the keys in the table, the k-mer's with frequency>=f can be extracted efficiently (using the length of each list and the table's constant-time access to an item). Once the frequent k-mers are found, the point-x mutations of a given frequent k-mer can be found in (nearly) constant time by simply searching for the mutations in the table. More details on the running time are provided below.

There are two viable approaches for building the hash table. The first uses Python dictionaries and the second is to construct something like a "list-of-lists" hash table. The advantage of the latter is that all n-k k-mers (k*(n-k) characters) do not have to be stored (n is the length of S), so there are considerable memory savings. The advantages of the dictionary approach are every other aspect of the problem. The dictionary approach is described here. A list-of-list implementation is discussed (in less detail) at the bottom of this document.

The analysis of the running time is divided into three parts – 1) the cost of constructing the hash-table, 2) the cost of constructing L1 and L2, and 3) the cost of counting the point-x mutations of the k-mers in L1.

1) A loop iterates through the n-k k-mers and "searches" for each in the dictionary (O(1) time). If not found, the k-mer is inserted into the dictionary (O(1) time) unless the k-mer is already in the dictionary in which case there is a search+append (O(1) time) Overall, the asymptotic running time is O(n-k), and the leading-order running time strongly depends on the cost of the searches and insertions in the dictionary.

2) We iterate through the items in the dictionary (worst-case min(n-k,4^k) iterations, O(1) time for lookup of an item) and there is one comparison with f per item (2 operations). If the k-mer is "frequent" there are appends to L1 and L2 (best case, 2 operations). Let's say there are F frequent k-mers, then the running time will be (C+1)min(n-k,4^k) + 2F where C is the cost of looking up an item in the dictionary, and in the worst case, when $4^k > n - k$, we have a worst-case asymptotic time of O(n-k).

3) For each of F frequent k-mers, we: ii) 1st append 0 to L3, ii) identify which 3 characters correspond to mutations (3 comparisons), iii) assemble the strings corresponding to each mutation (worst case 3k insertions), iv) search for each mutation in table (3C operations), v) if found, lookup count and increment count in L3 (3(2+C) operations). The estimated cost is then 3F(3 + k + 2C) + lower-order terms. The asymptotic cost (and leading-order cost) depends on the magnitudes of k and C.

The total cost then depends strongly on F, the number of frequent k-mers and k, however in practice, it is the 1st step which is most expensive, and the 2nd step is typically much faster than the others as the constant in its leading-order

constant is relatively small. For example, if k=3, there are only 64 possible k-mers, so even with f=1, the maximum value of F is 64 and the total cost of steps 2 and 3 is very small compared to the cost of step 1. For larger k, there is closer competition. With k=100 and f=1, tests with a n=1e6 sequence indicate that the cost of 1) and 3) are then comparable.

Part 2. (12 pts)

1) Complete the function *nsearch* in *p2_dev.py* so that it efficiently searches for *all* occurrences of a target within N partially sorted lists of numbers of length M. Specifically, the first P numbers in each list are sorted in ascending order. If the target is found in one or more lists, the function should determine which list(s) the target is in and the location within each list where the target is located. If the target is not found, simply return an empty list. See the function documentation for more details on the structure of the function input and output. You may assume that N, P, and M-P are all large.

Ans: The basic approach is to use binary search for the sorted portion of each sublist and linear search for the unsorted portion. Binary search has to be modified to account for the possibility of repeat occurrences of the target, and this modification takes the form of an additional linear search for the sub-interval in which binary search locates the target. The full code for *nsearch* is below:

```
def nsearch(L,P,target):
    """Input:
    L: list containing *N* sub-lists of length M. Each sub-list
    contains M numbers (floats or ints), and the first P elements
    of each sub-list can be assumed to have been sorted in
    ascending order (assume that P<M). L[i][:p] contains the sorted elements in
    the i+1th sub-list of L
    P: The first P elements in each sub-list of L are assumed
    to be sorted in ascending order
    target: the number to be searched for in L

    Output:
    Lout: A list consisting of Q 2-element sub-lists where Q is the number of
    times target occurs in L. Each sub-list should contain 1) the index of
    the sublist of L where target was found and 2) the index within the sublist
    where the target was found. So, Lout = [[0,5],[0,6],[1,3]] indicates
    that the target can be found at L[0][5],L[0][6],L[1][3]. If target
    is not found in L, simply return an empty list (as in the code below)
    """

    Lout=[]

    #iterate through sublists in L
    for i,Lsub in enumerate(L):


        #run binary search on Lsub[:p]-----------------
        #Set initial start and end indices for full sublist
        istart = 0
        iend = P-1

        #Iterate and contract to "active" portion of list
        while istart<=iend:

            imid = int(0.5*(istart+iend))

            if target==Lsub[imid]:
                Lout.append([i,imid])
                j=imid+1
                #Linear search within active portion of list--
                while target==Lsub[j]:
                    Lout.append([i,j])
```

```
                    j=j+1
                j = imid-1
                while target==Lsub[j]:
                    Lout.append([i,j])
                    j=j-1
                break
                #-------------------------
            elif target < Lsub[imid]:
                iend = imid-1
            else:
                istart = imid+1
        #Finished binary search
        #----------------------

        #Run linear search on Lsub[P:]-----------
        j = P
        for x in Lsub[P:]:
            if x==target: Lout.append([i,j])
            j=j+1
        #Finished linear search-------------

    return Lout
```

2) Carefully analyze the running time of *nsearch*. Your analysis should include: a) a concise description of your algorithm b) a clear and concise discussion of the running time and how it depends on the input, c) a concise explanation of if/why your algorithm can be considered to be efficient, d) one or more figures clearly illustrating key trends in the actual running time of the function, and e) a description of and explanation of the trends shown in the figure(s). The code that generates your figures should be placed in *nsearch_time*. Place your discussion in the docstring of *nsearch_time* as indicated in the template file. You should save your figure(s) as *png* files following the naming convention *fig1.png*, *fig2.png*, … and submit the figures along with your codes.

Ans:

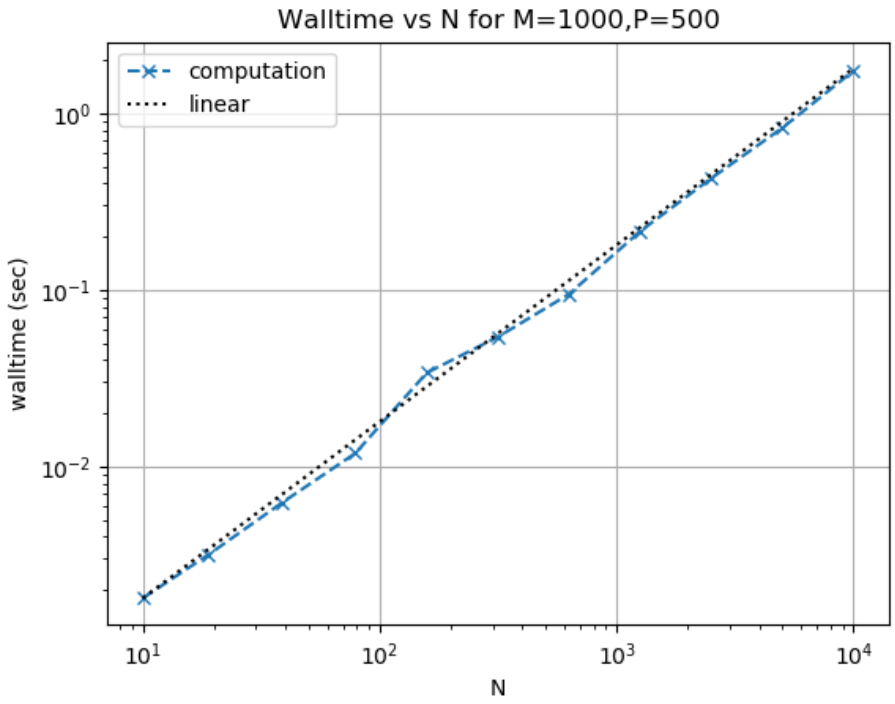
a) The algorithm has been described above.

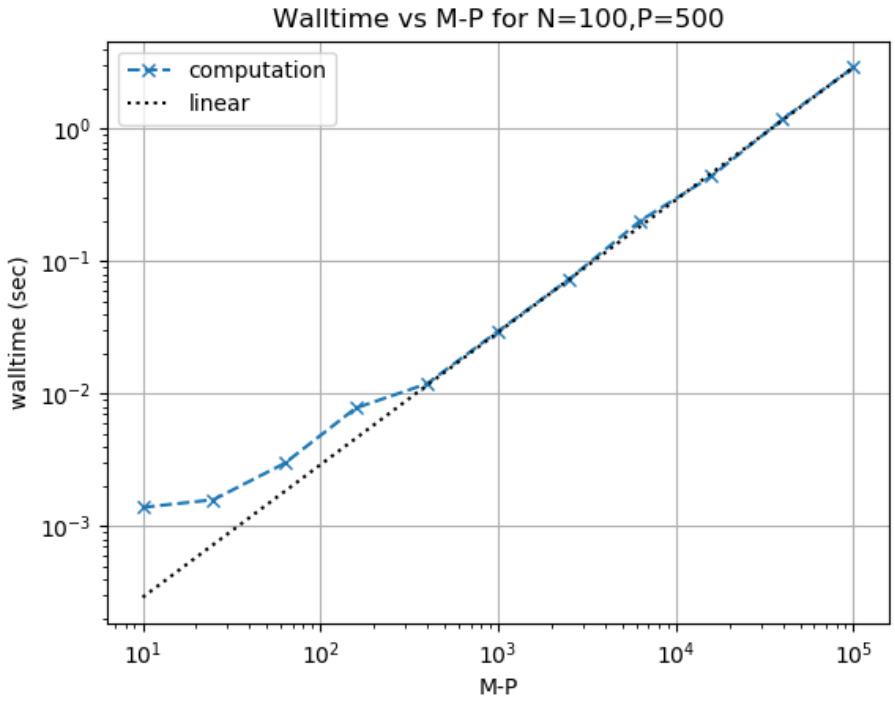
b) The worst-case running time for linear search in the unsorted portion of a sublist is approximately 4(M-P) which corresponds to incrementing 2 integers, 1 comparison, and an append (if the target is found). Binary search is more complicated. If the target is not in the sublist, then the cost is approximately 6(ceiling(log2(P))+1) where “ceiling” indicates rounding up to the nearest integer and the “6” corresponds to the calculation of imid (3 operations), 2 comparisons, and the adjustment of istart or iend, though it the precise number is not important. There is an additional cost for each sublist updating i and Lsub and setting istart and iend (4 operations). For N sublists then, the approximate total running time is: 4N(M-P) + 6Nlog2(P)+10N For large M-P, P, and N, the asymptotic running time will be O(N(M-P)) unless log2(P)>>(M-P) but there is not enough memory on my laptop to reach this regime while also keeping M-P large!

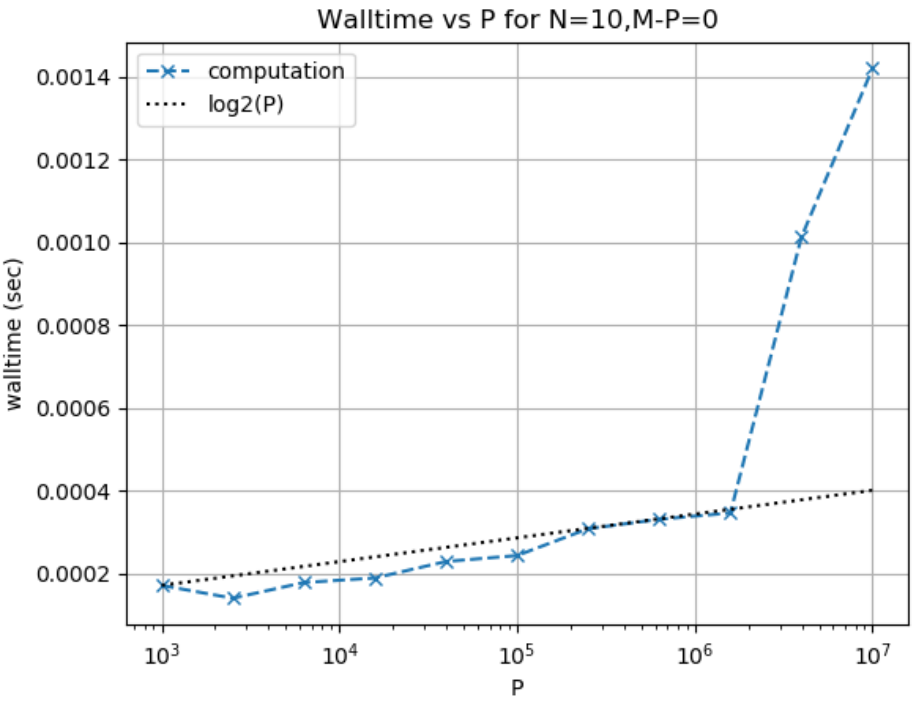
c) The algorithm is efficient only in its use of binary search for the sorted portions of the sublists. Otherwise, it is important to first sort the unsorted portion of the sublists (or to merge the sorted portions) as the cost of the sort is greater than simply using linear search without sorting.

d) and e) The three figures below illustrate how the walltime depends on the input size with two parameters held fixed and one varying. The simplest case is varying N – we expect a clear linear trend – and that is seen in the figure. In the second figure, M is varied with P and N held fixed. Again a linear trend is expected, and for sufficiently large (M-P) a clear linear trend is indeed seen in the figure. At smaller (M-P), the cost of the binary search becomes comparable, which results in a departure from the plotted trend due to the use of a log-log plot. The 3rd figure shows the cost of the binary search portion of the algorithm. As mentioned above, binary search is so fast relative to linear search that it can be difficult to “find” the expected log2 trend. So, I have just set M=P, and, up to about P=1e6, we do see this log2 behavior, however there is a rapid deterioration in performance for larger P due to memory limitations (and the large number of applications open on my laptop). Note that a better approach would have been to use something like np.polyfit to construct actual fits to the data. There are also other avenues that could have been

explored by, say, varying P and M simultaneously. The code used to generate the figures can be found below the figures.








```
def nsearch_time(Nsize=11,Msize=11,Psize=11):
    """Analyze the running time of nsearch.
    Add input/output as needed, add a call to this function below to generate
    the figures you are submitting with your codes.
    """
    import numpy as np
    import matplotlib.pyplot as plt
    from time import time

    #Vary N, M,P fixed--------------
    Narray = np.logspace(1,4,Nsize,dtype=int)
    M=1000
    P=500
    dtarray_N = np.zeros(Nsize)
    for i,N in enumerate(Narray):
        print("i,N=",i,N)
        Linput = generate_input(N,M,P)
        t1=time()
        Lout = nsearch(Linput,P,-1)
        t2 = time()
        dtarray_N[i] = t2-t1

    #Display results and compare to expected linear trend
    plt.figure()
    plt.loglog(Narray,dtarray_N,'x--')
    x = np.logspace(1,4,1001)
    plt.plot(x,x/x[0]*dtarray_N[0],'k:')
    plt.xlabel('N')
    plt.ylabel('walltime (sec)')
    plt.title('Walltime vs N for M=%d,P=%d' %(M,P))
    plt.grid()
    plt.legend(('computation','linear'))
    #---------------------------------------------

    #Vary M-P, N,P fixed----------------------
    Marray = np.logspace(1,5,Msize,dtype=int)
    N=100
    P=500
    Marray += P
```

```
    dtarray_M = np.zeros(Msize)
    for i,M in enumerate(Marray):
        print("i,M=",i,M)
        Linput = generate_input(N,M,P)
        t1=time()
        Lout = nsearch(Linput,P,-1)
        t2 = time()
        dtarray_M[i]= t2-t1

    #Display results and compare to expected linear trend
    plt.figure()
    plt.loglog(Marray-P,dtarray_M,'x--')
    x = np.logspace(1,5,1001)
    plt.plot(x,x/x[-1]*dtarray_M[-1],'k:')
    plt.xlabel('M-P')
    plt.ylabel('walltime (sec)')
    plt.title('Walltime vs M-P for N=%d,P=%d' %(N,P))
    plt.grid()
    plt.legend(('computation','linear'))
    #-----------------------------

    #Vary P, M-P fixed
    Parray = np.logspace(3,7,Psize,dtype=int)
    N=10
    dtarray_P = np.zeros(Psize)
    for i,P in enumerate(Parray):
        M = P
        print("i,P,M=",i,P,M)
        Linput = generate_input(N,M,P)
        t1=time()
        Lout = nsearch(Linput,P,-1)
        t2 = time()
        dtarray_P[i]= t2-t1

    #Display results and compare to expected logarithmic trend
    plt.figure()
    plt.semilogx(Parray,dtarray_P,'x--')
    x = np.logspace(3,7,1001)
    plt.semilogx(x,np.log2(x)/np.log2(x[0])*dtarray_P[0],'k:')
    plt.xlabel('P')
    plt.ylabel('walltime (sec)')
    plt.title('Walltime vs P for N=%d,M-P=%d' %(N,M-P))
    plt.grid()
    plt.legend(('computation','log2(P)'))


    return Narray,dtarray_N,Marray,dtarray_M,Parray,dtarray_P #Modify as needed
```

List-of-list hash table for part 1

The code is below. A rolling hash is used to construct a hash table which is a list of lists. The sub-lists contain the locations of the k-mers in S and a running count of the k-mer's frequency. A large prime number, q, is used to ensure that that the size of the hash table is constrained.

The basic approach for building the table is as follows:

1. Convert string into base-4 number (list of integers between 0 and 3) – cost(O(n-k))
2. Iterate through this number, considering each "k-mer" individually (n-k iterations)

   i) Check if hash is in list (1 comparison), if not, insert location of k-mer into table (1 append), if already in list, search through sublist to verify a "true match" (~4 operations/iteration of search) and if there is not a true match, 1 append.

ii. Update hash (~6 operations)

Then, to extract the frequent k-mers:

Iterate through all items in list-of-lists and each iteration:

1. check for sublists and iterate through them if present
2. find counts which are greater than f
3. append locations and k-mers to L1 and L2

The outer loop requires q iterations, but most operations will take place min(4^k,n-k) times, so the worst-case asymptotic time is O(n-k) – see discussion of dictionary-based implementation above for further details.

For each of F frequent k-mers, we: ii) 1st append 0 to L3, ii) identify which 3 characters correspond to mutations (3 comparisons), iii) assemble the strings corresponding to each mutation (worst case 3k insertions), iv) search for each mutation in table (3 operations), v) if found, lookup count and increment count in L3 (>3 operations). The estimated cost is then 3F(2 + k + 3) + lower-order terms assuming that the number of hash collisions is small. The asymptotic cost (and leading-order cost) depends on the magnitudes of k and C. The probability of a hash collision depends on k, n and q. S contains a maximum of 4^k k-mers so we can expect ~ min(n-k,4^k)/q collisions.

The asymptotic costs of each of these steps is similar to the dictionary implementation discussed above assuming that there are few hash collisions, however the walltime for the dictionary approach will tend to be shorter since Python dictionaries are built using highly-optimized C code highly-optimzed. On the other hand, dictionaries are memory-intensive, and this list-of-list implementation requires O(N-k) memory rather than the O(k(N-k)) required by the dictionary, so when k and N are both large with N>>k, a loss of performance due to excessive memory use will be observed for the dictionary method before it is seen for the list-of-list approach.

```
def ksearch2(S,k,f,x):
    """
    Search for frequently-occurring k-mers within a DNA sequence
    and find the number of point-x mutations for each frequently
    occurring k-mer.
    Input:
    S: A string consisting of A,C,G, and Ts
    k: The size of k-mer to search for (an integer)
    f: frequency parameter -- the search should identify k-mers which
    occur at least f times in S
    x: the location within each frequently-occurring k-mer where
    point-x mutations will differ.

    Output:
    L1: list containing the strings corresponding to the frequently-occurring k-mers
    L2: list containing the locations of the frequent k-mers
    L3: list containing the number of point-x mutations for each k-mer in L1.

    Discussion: Add analysis here
    """
    n=len(S)
    #Below k=10 or so, no need to hash
    prime = 9999889

    #Convert string to base4

    L = char2base4(S)

    #Initialize hash table
    T = [[] for i in range(prime)]

    #Precompute factor for rolling hash
    bm = 4**k%prime

    #Compute 1st hash
```

```
    ind=0
    hi = heval(L[ind:ind+k],4,prime)

    #---Build Hash Table--------
    #Iterate through base4 number
    for ind in range(0,n-k+1):
        #Check for hash collision
        if len(T[hi])==0:
            #if new, create new sublist containing: [index]
            T[hi].append([ind])
        else:
            #print("collision, ind=",ind,"hi=",hi)
            found=0
            for j,l in enumerate(T[hi]):
                Lind = l[0]
                if L[Lind:Lind+k]==L[ind:ind+k]:
                    T[hi][j].append(ind)
                    found=1
                    break
            if found==0: T[hi].append([ind])
        #if not, sequentially compare and append new sublist if new
        #or append to old sublist if not

        #update hash
        if ind<n-k: hi = (4*hi - (L[ind])*bm + (L[ind+k])) % prime

    L1=[]
    L2=[]

    #---Extract frequently-occurring k-mers----
    for t in T:
        if len(t)>0:
            for l in t:
                if len(l)>=f:
                    ind=l[0]
                    L1.append(S[ind:ind+k])
                    L2.append(l)

    #Find point-x mutations
    L3=[]
    for l in L1:
        L3.append(0)
        for c in "ACGT": #iterate through point mutations
            if c != l[x]:
                l2 = l[:x] + c + l[x+1:]
                l3 = char2base4(l2)
                #compute hash of mutation
                hi = heval(l3,4,prime)

                #check if hash is present in table
                if len(T[hi])>0:
                    #scan through table entry and find exact match
                    for l4 in T[hi]:
                        Lind = l4[0]
                        if S[Lind:Lind+k]==l2:
                            L3[-1]+=len(l4)
                            break

    return L1,L2,L3

def char2base4(S):
    """Convert gene test_sequence
    string to list of ints between 0 and 4
    """
    c2b = {}
```

```
    c2b['A']=0
    c2b['C']=1
    c2b['G']=2
    c2b['T']=3
    L=[]
    for s in S:
        L.append(c2b[s])
    return L



def heval(L,Base,Prime):
    """Convert list L to base-10 number mod Prime
    where Base specifies the base of L
    """
    f = 0
    for l in L[:-1]:
        f = Base*(l+f)

    h = (f + (L[-1])) % Prime
    return h
```