Scientific Computation Project 1 solution

November 15, 2024

Question 1

1(a)

Method 1 uses linear search ($\mathcal{O}(n)$ cost) to find the location of a target in the list, and m linear searches will have $\mathcal{O}(mn)$ cost in both the average and worst cases. For Method 2, use `flag=True` on the first call and `False` on the subsequent $m-1$ calls. This approach first creates a new version of the list which also contains the list element indices ($\mathcal{O}(n)$ cost), then applies merge sort to the list ($\mathcal{O}(n \log n)$ worst-case and average-case costs), and then applies binary search m times to the sorted list ($\mathcal{O}(m(\log n)$ worst-case and average-case costs). The cost of creating the new list is small compared to the cost of merge cost for large n, so the big-O cost of method 2 is $\mathcal{O}((m+n)\log n)$. We can expect method 2 to be faster than method 1 if m is sufficiently large.

1(b)

Here, we will just consider the worst-case cost which occurs when none of the m targets are in the initial list. The average-case cost trends will be similar, but require averaging over several repetitions. Let $c_i(m,n)$ be the estimated cost of method i, and let $t_i(m,n)$ be the wall time required by method i. For method 1, we have $c_1 = \kappa_1 mn$ where κ is a constant that depends on the number of operations (from lecture, we expect $\kappa \approx 3$). For method 2, $c_2 = \mu_1 n \log_2 n + \mu_2 n + \mu_3 m(\log_2 n + 1)$ and μ_1, μ_2, and μ_3 depend on the detailed number of operations (the 2nd term is the cost of creating the list which contains the element indices). From lecture and lab 1, we expect $\mu_1, \mu_3 \approx 11$. I will assume that n is large enough so that $c_2 \approx \mu_1 n \log_2 n + \mu_3 m \log_2 n$. Figure 1(a) shows t_2/m vs. n for a few different values of n. From the cost estimate, we expect a linear dependence on n and no dependence on m, and that is what we see (the least-squares estimate of the slope is very close to 1). For method 2, note that $c_2/m \approx (\mu_1 n/m + \mu_3)\log_2 n$. We again vary m and n, but now it is varied so that the ratio, m/n is held fixed to a specified value (0.2, 0.3, or 0.4). Then we expect t_2/m to increase logarithmically with n with the slope (on a linear-log plot) increasing as m/n decreases (since then $(\mu_1 n/m + \mu_3)$ increases). Figure 1(b) confirms these expectations. Each curve does appear to follow a logarithmic trend (least-squares fits are shown for reference), and the slopes do indeed increase as m/n decreases. Next, consider the wall time ratio, t_2/t_1, which is shown in figure 1(c). As noted in question (a), for a given n, we expect this ratio to decrease as m is increased since the cost of merge sort is increasingly offset by cost savings from replacing linear search with binary search. For all but the smallest value of n, this general trend is clearly seen in the figure. Based on our cost estimates, we have $r = c_2/c_1 \approx \log_2 n/\kappa\,(\mu_1/m + \mu_3/n)$. Consider the behavior of r for a given m with n varying. There will then be competition between the two terms in parentheses. For sufficiently large n, the first term will dominate, and we expect a logarithmic increase in r and t_2/t_1. The larger m is, the larger n needs to be to see this domination which is why we do not really see a logarithmic increase for the largest m in figure 1(c). Why does t_2/t_1 decrease with n in some parts of the figure? This is due to competition between $\log_2 n\mu_1/m$ and $\log_2 n/\kappa(\mu_2/n)$ The first term increases with n while the second term decreases. Then, if m is sufficiently large relative to n, this decreasing term dominates, and the wall time ratio decreases with n. We can also directly compare our estimate for r with the measured wall times if we estimate κ, μ_1, and μ_2. First, assume $\mu_1 \approx \mu_2 \approx 11$ and then if $\kappa \approx 3$, $r \approx 4\log_2 n\,(1/m + 1/n)$. A little experimentation shows that 5 works better than 4, and the resulting approximation is shown in figure 1(d). While there are tangible quantitative differences between this figure and figure 1(c), the strong qualitative similarities are convincing, and our final approximation for the cost ratio captures all important trends.

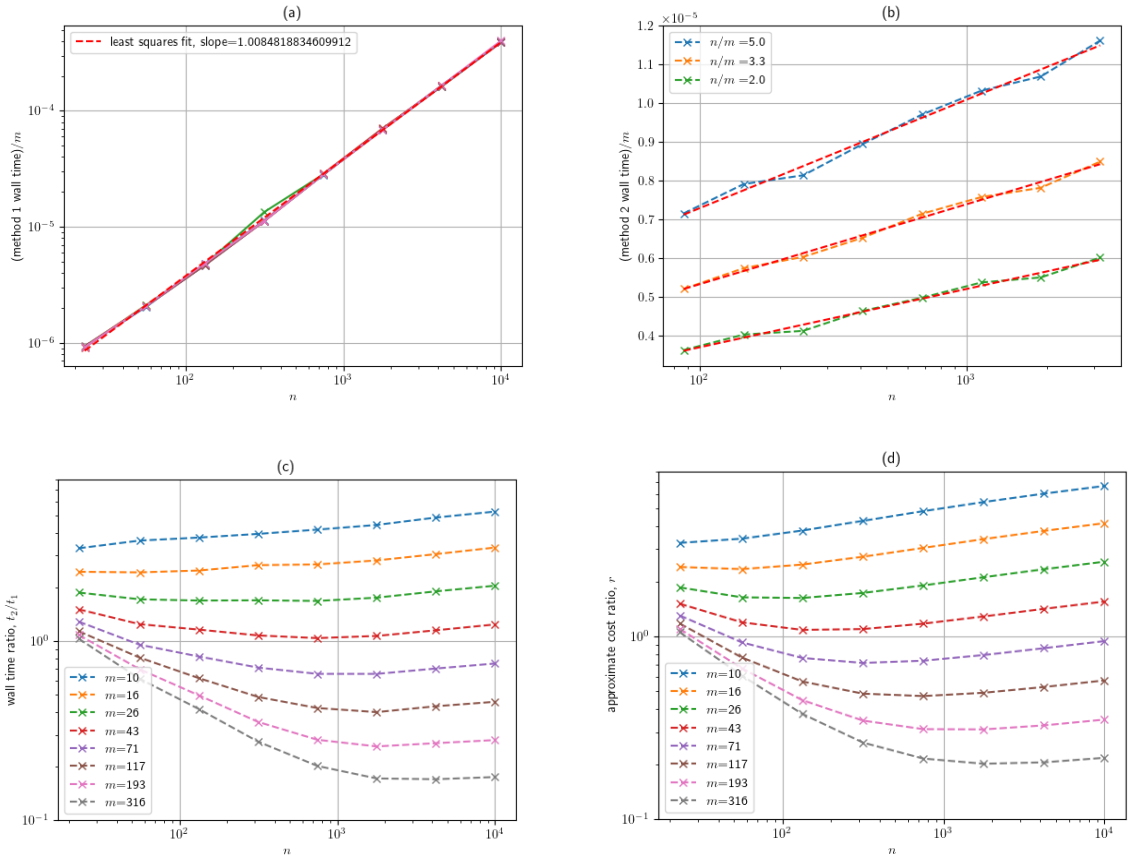

Figure 1: (a) t_1/m vs. n, (b) t_2/m vs. n, (c) wall-time ratio t_2/t_1 vs. n, (d) approximate cost ratio, c_2/c_1 vs. n

Question 2

2(b)

Part 2

The general approach is to use a version of the Rabin-Karp algorithm with pattern hashes stored in a dictionary. Hashes for patterns are constructed by first converting all of our strings to base-9 numbers stored in lists (using *char2base9*). Then a length-m pattern is viewed as a 2 x m matrix. A base-10 hash is constructed by "unrolling" the matrix one column at a time. For example, with $m = 3$ and $S_{i,j}$ corresponding to the jth base-9 integer in the ith input sequence, the hash for the pair of patterns $[S_{1,j}, S_{1,j+1}, S_{1,j+2}]$ and $[S_{2,j}, S_{2,j+1}, S_{2,j+2}]$ is:

$$H_j = 9^5 S_{1,j} + 9^4 S_{2,j} + 9^3 S_{1,j+1} + 9^2 S_{2,j+1} + 9 S_{1,j+2} + S_{2,j+2}.$$

The hash update is managed by precomputing hashes of columns, $c_j = 0S_{1,j} + S_{2,j}$. Then, the hash update is: $H_{j+1} = 9H_j - 9^{2m}c_j + c_{j+m}$. The hashes of the pattern-pairs in L_p are precomputed and stored in a dictionary. Then, after each hash update, there is a lookup of the new hash in this dictionary. If it is found, F is updated appropriately.

The cost of converting all of the input to base-9 is $O(n + mp)$ as the cost of converting one character requires direct access in a string, a dictionary lookup, and a list append which are all $O(1)$ operations. Examining *heval*2, we see that the computation of a hash of a 2 x m base-9 matrix requires approximately $9m + C_1$ additions and $9m + C_2$ multiplications where C_1 and C_2 are small positive integers. Each of these hashes is inserted into a dictionary which is $O(1)$ so the cost of creating the pattern dictionary is $O(ml)$. There are a few additional constant time operations to set/initialize various parameters which we will neglect here.

Now consider the cost of searching through the pair of base-9 sequences. The initial hash calculation requires $O(m)$ operations as discussed above. There are $n - m$ hash updates which each require 2 multiplications and 2 additions, so the hash updates require $O(n)$ operations in total since $n \gg m$. With the same reasoning, the total cost of hash calculations for the pair of input sequences is $O(n)$. With each hash calculation, there is a lookup of the hash in the pattern dictionary, and if there is a match, there is an append to F. These are all constant time operations so, the total cost of "processing" the pair of input sequences remains $O(n)$. Finally, note that the computation of c_j requires n additions and multiplications, and the total cost of the overall routine is estimated as $O(n + ml)$. With regards to asymptotic running time, this is close to the best that we could hope for without making assumptions about the input (e.g. assumptions on how many matches there are or how similar the two input sequences are) since each character in A1 or A2 needs to be examined along with each character in L. Naive pattern search, for example, has $\mathcal{O}(ln)$ best-case cost. Note that there is no need to reduce the hashes modulo a large prime as we assume that there is no performance loss associated with large integers. This algorithm is insensitive to the number of matches which only affects the number of appends to F. Any algorithm will require these appends.

Code

```
def part1_test(inputs=None):
    """Part 1, question 2: investigate trends in wall times of methods 1 and 2.
    Use the variables inputs and outputs if/as needed.
    You may import modules within this function as needed, please do not import
    modules elsewhere without permission.
    """
    import numpy as np
    from time import time
    import matplotlib.pyplot as plt
    plt.rcParams['text.usetex'] = True

    #Collect timing information for two methods

    #Set range of values of m and n
    nsize = 9
    msize = 8
    reps = 200 #average time computed over reps repetitions
    nvalues = np.logspace(1,4,nsize,dtype=int)
    mvalues = np.logspace(1,2.5,msize,dtype=int)

    t1values = np.zeros((nsize,msize))
    t2values = np.zeros((nsize,msize))

    #Collect timing results
    for i in range(nsize):
        n = nvalues[i]
        print(i,n)
        L1 = list(np.random.randint(0,2*10**5,n))
        for j in range(msize):
            m = mvalues[j]
            t1 = time()
            for r in range(reps):
                for k in range(m):
                    out = method1(L1,-1)
```

```
        t2 = time()
        t1values[i,j]=(t2-t1)/reps

        t1 = time()
        for r in range(reps):
            out,L1n = method2(L1,-1,flag=True)
            for k in range(m):
                out = method2(L1n,-1,flag=False)
        t2 = time()
        t2values[i,j] = (t2-t1)/reps

#discard 1st unreliable result
nvalues = nvalues[1:]
t1values = t1values[1:,:]
t2values = t2values[1:,:]

#display results
#figure 1(a)
y = t1values/mvalues
plt.figure()
plt.loglog(nvalues,t1values/mvalues,'x-')
plt.grid()
plt.xlabel(r'$n$')
plt.ylabel(r'(method 1 wall time)/$m$')
p1,p2 = np.polyfit(np.log(nvalues),np.log(y[:,-1]),1)
plt.plot(nvalues,np.exp(p2)*nvalues**p1,'r--',label=f'least squares fit, slope={p1}')
plt.title('(a)')
plt.legend()

#figure 1(c)
plt.figure()
for i,m in enumerate(mvalues):
    plt.loglog(nvalues,t2values[:,i]/t1values[:,i],'x--',label=r'$m$=%d'%m)
plt.grid()
plt.xlabel(r'$n$')
plt.ylabel(r'wall time ratio, $t_2/t_1$')
plt.title('(c)')
plt.legend()

#figure 1(d)
plt.figure()
for i,m in enumerate(mvalues):
    plt.loglog(nvalues,3*np.log2(nvalues)*(1/m+1/nvalues),'x--',label=r'$m$=%d'%m)
plt.grid()
plt.xlabel(r'$n$')
plt.ylabel(r'approximate cost ratio, $r$')
plt.title('(d)')
plt.legend()


#Timing tests to assess wall time of method2 vs n
nsize2 = 10
nvaluesb = np.logspace(1.5,3.5,nsize2,dtype=int)
nmvalues = np.array([0.2,0.3,0.5])
t2valuesb = np.zeros((nsize2,len(nmvalues)))
for i in range(nsize2):
    n = nvaluesb[i]
    print(i,n)
```

```
            L1 = list(np.random.randint(0,2*10**5,n))
            for j,nm in enumerate(nmvalues):
                m = int(n*nm)
                t1 = time()
                for r in range(reps):
                    out,L1n = method2(L1,-1,flag=True)
                    for k in range(m):
                        out = method2(L1n,-1,flag=False)
                t2 = time()
                t2valuesb[i,j] = (t2-t1)/reps

    nvaluesb = nvaluesb[1:]
    t2valuesb = t2valuesb[1:,:]
    #compute least-squares fit and create figure 1(c)
    plt.figure()
    p1vals = []
    p2vals = []
    for j,nm in enumerate(nmvalues):
        mvaluesb = nm*nvaluesb
        y = t2valuesb[:,j]/mvaluesb
        plt.semilogx(nvaluesb,y,'x--',label=r'$n/m=$%2.1f'%(1/nm))
        p1,p2 = np.polyfit(np.log(nvaluesb),y,1)
        plt.plot(nvaluesb,p1*np.log(nvaluesb)+p2,'r--')#,label='least squares fit, slope=%2.1f, £n/m
        p1vals.append(p1)
        p2vals.append(p2)

    plt.legend()
    plt.xlabel(r'$n$')
    plt.ylabel(r'(method 2 wall time)/$m$')
    plt.title('(c)')
    plt.grid()

    outputs=nvalues,mvalues,t1values,t2values,nvaluesb,t2valuesb
    return outputs

def part2(A1,A2,L):
    """Part 2: Complete function to find amino acid patterns in
    amino acid sequences, A1 and A2
    Input:
        A1,A2: Length-n strings corresponding to amino acid sequences
        L: List of l sub-lists. Each sub-list contains 2 length-m strings. Each string corresponds t
        sequence
    Output:
        F: List of lists containing locations of amino-acid sequence pairs in A1 and A2.
        F[i] should be a list of integers containing all locations in A1 and A2 at
        which the amino acid sequence pair stored in L[i] occur in the same place.
    """

    #use/modify the code below as needed
    n = len(A1) #A2 should be same length
    l = len(L)
    m = len(L[0][0])
    F = [[] for i in range(l)]

    base = 9
    #Compute hashes of AA pairs and store in dictionary
    PDict = {}
    for i,pattern in enumerate(L):
```

```
            p1,p2 = char2base9(pattern[0]),char2base9(pattern[1])

            h,_,_,_ = heval2(p1,p2,base)
            if h in PDict:
                PDict[h].append(i)
            else:
                PDict[h] = [i]

    #Convert input sequence to list of base 4 integers
    A1 = char2base9(A1)
    A2 = char2base9(A2)

    #Convert adjacent sequences into single sequence of "pre-hashes"
    L2 = []
    for j in range(n):
        L2.append(base*A1[j]+A2[j])

    b2  = base*base
    b2m = b2**(m)
    hI0 = 0


    #Iterate across pair of sequences, updating hash, and checking
    #if hash is in PatternDict (and updating locations as appropriate)

    for i in range(n-m+1):
        if i==0: #first hash for a pair of sequences
            #first pair of sequences
            hI,c1,r1,r2 = heval2(A1[:m],A2[:m],base)

            if hI in PDict:
                #print("Found pattern: hI=",hI)
                for ind in PDict[hI]:
                    F[ind].append(i)
            hI0 = hI
        else:
            hI = b2*hI - b2m*L2[i-1] + L2[i-1+m]

            if hI in PDict:
                for ind in PDict[hI]:
                    F[ind].append(i)


    return F


def heval(L,Base):
    """Hash of one "row" of 2 x k array of  base-x integers with x=Base
    """
    Base2 = Base*Base
    f = 0
    for l in L[:-1]:
        f = Base2*(int(l)+f)

    h = (f + int(L[-1]))
    return h

def heval2(L1,L2,Base):
```

```
    """heval modified to compute hash from two equal length lists
    combined into a single list
    The lists are viewed as a 2 x m array and the hash is computed by
    unrolling the array one column at a time. c1 corresponds to the
    first column of this array and r1 and r2 correspond to the two rows
    """
    r1 = 0
    r2 = 0
    c1 = Base*L1[0]+L2[0]
    Base2 = Base*Base
    for l1,l2 in zip(L1[:-1],L2[:-1]):
        r1 = Base2*(int(l1)+r1)
        r2 = Base2*(int(l2)+r2)

    r1 = Base*(r1 + int(L1[-1]))
    r2 = r2 + int(L2[-1])
    h = r1 + r2

    return h,c1,r1,r2

def char2base9(S):
    """Convert AA test_sequence
    string to list of ints
    """
    c2b = {}
    for i,s in enumerate('abcdefghi'):
        c2b[s] = i
    L=[]
    for s in S:
        L.append(c2b[s])
    return L


if __name__=='__main__':
    A1 = 'afgabciiabcb'
    A2 = 'feadefdddefe'
    L = [['abc', 'def' ], ['afg', 'fea']]

#----------------- End code for Part 2 -----------------#
```