CS532 Homework 4
Archana Machireddy

Question 1

At each location we have three choices, moving right, down or diagonally right down. It is like search on a tree where each node has 3 sub-children, and the sub-children may be repeated in multiple paths. For two arrays of size m and n, the maximum length of alignment is m+n. So the height of the tree is m+n. Going down each path would take O(m+n) time. At each node, the function needs to be evaluated three times. The evaluation triples with each addition to the input. So, the growth is $O(3^{m+n})$.

Question 2

```
def version1(a1,a2,x,y,seq):

    n = len(a1)
    m = len(a2)

    if x == n-1 and y == m-1:
        print(seq)
        return

    if x == n-1:
        version1(a1,a2,x,y+1,seq+'|')
        return

    if y == m-1:
        version1(a1,a2,x+1,y,seq+'-')
        return

    version1(a1,a2,x+1,y,seq+'-')
    version1(a1,a2,x,y+1,seq+'|')
    version1(a1,a2,x+1,y+1,seq+'\\')
```

Question 3

Size of Array	Time	t_n/t_{n-1}	t_n/t_1
1	0.000004		
2	0.000009	2.25	2.25

3	0.000026	2.888888889	6.50
4	0.000132	5.076923077	33.00
5	0.000684	5.181818182	171.00
6	0.002767	4.045321637	691.75
7	0.013062	4.720636068	3265.50
8	0.064444	4.933700812	16111.00
9	0.349355	5.421063249	87338.75
10	1.957277	5.602544689	489319.25

The rate at which the time is increasing as the array size is growing is exponential. It takes 2.25 times more time when the size is doubled. From then on it almost takes 5 times more time for each increase in array size of 1. This increase is exponential

Question 4

```
def version2(a1,a2,x,y,seq,score):

    n = len(a1)
    m = len(a2)
    if x == n-1 and y == m-1:
        print '%-4i%-s' % (score,seq)
        return

    if x == n-1:
        version2(a1,a2,x,y+1,seq+'|',score+1)
        return

    if y == m-1:
        version2(a1,a2,x+1,y,seq+'-',score+1)
        return

    version2(a1,a2,x+1,y,seq+'-',score+1)
    version2(a1,a2,x,y+1,seq+'|',score+1)
    if a1[x+1] == a2[y+1]:
        version2(a1,a2,x+1,y+1,seq+'\\',score)
    else:
        version2(a1,a2,x+1,y+1,seq+'\\',score+1)
```

Question 5

```
def version3(a1,a2,x,y,seq,score):

    n = len(a1)
```

```
    m = len(a2)

    if x == n-1 and y == m-1:
        return (seq,0)

    if x == n-1:
        (cseq,cscor) = version3(a1,a2,x,y+1,seq+'|',score+1)
        return (cseq,cscor+1)

    if y == m-1:
        (cseq,cscor) = version3(a1,a2,x+1,y,seq+'-',score+1)
        return (cseq,cscor+1)

    scores = [float("inf"),float("inf"),float("inf")]
    seqs = ["", "", ""]
    seq11 = ['-','|','\\']

    seqs[0],scores[0] = version3(a1,a2,x+1,y,seq+'-',score+1)
    scores[0] += 1
    seqs[1],scores[1] = version3(a1,a2,x,y+1,seq+'|',score+1)
    scores[1] += 1
    if a1[x+1] != a2[y+1]:
        seqs[2],scores[2] = version3(a1,a2,x+1,y+1,seq+'\\',score+1)
        scores[2] += 1
    else:
        seqs[2],scores[2] = version3(a1,a2,x+1,y+1,seq+'\\',score)

    mn,idx = min( (scores[i],i) for i in xrange(len(scores)) )
    return (seqs[idx], mn)
```

Question 6

We are recalculating the lower paths repeatedly while going across the matrix. For example for arrays of length 5, for each path from the beginning the final path from (4,4) to (5,5) or (4,5) to (5,5) will be calculated multiple times, similarly multiple other sub-paths are common in many complete paths. At each cell if we store the best path to take from that cell to the last cell, we need not recompute the value for other paths in the future going down the same route. Instead, just append the path from that cell and add the cost.

Question 7

```
def version4(a1,a2,x,y,seq,score):

    n = len(a1)
    m = len(a2)
```

```
    ### Initializing cache
    if x == -1 and y == -1:
        for i in range(-1,n):
            for j in range(-1,m):
                cache[(i,j)] = ("",-1)

    ### Checking if value updated in cache, if updated adding the path
to current path
    if cache[(x,y)][1] != -1:
        sequence = cache[(x,y)][0]
        sc = cache[(x,y)][1]
        return (seq + sequence,sc)

    if x == n-1 and y == m-1:
        cache[(x,y)] = ("",0)
        return ("",0)

    if x == n-1:
        (cseq,cscor) = version4(a1,a2,x,y+1,seq+'|',score+1)
        cache[(x,y)] = ('|'+cache[(x,y+1)][0],cscor+1)
        return (cseq,cscor+1)

    if y == m-1:
        (cseq,cscor) = version4(a1,a2,x+1,y,seq+'-',score+1)
        cache[(x,y)] = ('-'+cache[(x+1,y)][0],cscor+1)
        return (cseq,cscor+1)

    scores = [float("inf"),float("inf"),float("inf")]
    seqs = ["", "", ""]
    seq11 = ['-','|','\\']
    x1 = [x+1,x,x+1]
    y1 = [y,y+1,y+1]

    seqs[0],scores[0] = version4(a1,a2,x+1,y,seq+'-',score+1)
    scores[0] += 1
    seqs[1],scores[1] = version4(a1,a2,x,y+1,seq+'|',score+1)
    scores[1] += 1
    if a1[x+1] != a2[y+1]:
        seqs[2],scores[2] = version4(a1,a2,x+1,y+1,seq+'\\',score+1)
        scores[2] += 1
    else:
        seqs[2],scores[2] = version4(a1,a2,x+1,y+1,seq+'\\',score)

    mn,idx = min((scores[i],i) for i in xrange(len(scores)))
    cache[(x,y)] = (seq11[idx]+cache[(x1[idx],y1[idx])][0], mn)
    return (seqs[idx], mn)
```

Question 8

The contents of the input array does not matter as the code goes through the entire sequences and stores the path and cost from a given point on towards the end. Irrespective of whether the arrays are already aligned or are completely different, the code calculates cost for the best move from every location to the next. For two strings of length m and n, there are mn sub-problems, and it takes same amount of work for each sub-problem. So, the best and worst-case performances are the same.

Question 9

Array size	time	constant (c)
10	0.000946	9.46E-06
20	0.003642	9.11E-06
30	0.007816	8.68E-06
40	0.01231	7.69E-06
50	0.021041	8.42E-06
60	0.03214	8.93E-06
70	0.04351	8.88E-06
80	0.057149	8.93E-06
90	0.068416	8.45E-06
100	0.083967	8.40E-06
110	0.103691	8.57E-06
120	0.129382	8.98E-06
130	0.141506	8.37E-06
140	0.175151	8.94E-06
150	0.22597	1.00E-05
160	0.238096	9.30E-06
170	0.260335	9.01E-06
180	0.293821	9.07E-06
190	0.346565	9.60E-06
200	0.358424	8.96E-06
210	0.415783	9.43E-06
220	0.459236	9.49E-06
230	0.474166	8.96E-06
240	0.541047	9.39E-06
250	0.667131	1.07E-05
260	0.714037	1.06E-05
270	0.778652	1.07E-05
280	0.843566	1.08E-05
290	0.937682	1.11E-05
300	0.992464	1.10E-05
310	1.092573	1.14E-05

The constant is similar and is between 0.000009 and 0.000018

320	1.17057	1.14E-05
330	1.245066	1.14E-05
340	1.331177	1.15E-05
350	1.569991	1.28E-05
360	1.686239	1.30E-05
370	1.811717	1.32E-05
380	1.935963	1.34E-05
390	2.283281	1.50E-05
400	2.515518	1.57E-05
410	2.480374	1.48E-05
420	3.03453	1.72E-05
430	6.343956	3.43E-05
440	4.582138	2.37E-05
450	3.548734	1.75E-05
460	3.784996	1.79E-05
470	3.875978	1.75E-05
480	4.177474	1.81E-05
490	4.469089	1.86E-05

Question 10

The code computes the cost and the best subsequence at each location (x,y). For each call of version 4 say was take time T(k) to run it. To compute the best score at a point it calls the function 3 times, taking the min takes constant time. There are 3 * m * n calls to the sub-problem. This results in O(mn) of the algorithm.

Question 11

In the bottom-up implementation, value of the cells on top-left are filled first and further evaluated bottom-right. From the starting of the sequence the best direction for the sequence to take at each step is calculated. So when you reach the bottom right corner, the best cost to traverse the two arrays will be the cost of this cell. Smaller parts of the problem in the beginning are solved first and are extended to solve the full problem. Each cell has optimal cost to traverse up to that point.

In the top-down approach, after solving one sub-problem we store its value, and subsequent calls will check the stored values before re-computing the values in that path. Here we are filling the values of bottom-right of the matrix first and keep computing values to top-left. Here instead of repeatedly calculating values of the sub-problems, we store them and look-up. Each cell has optimal cost from that point on to the end.