Matt Dumford
Anthony Phelps

CS 401 Homework 3

3) a. For each of the four endpoints of the two lines, perform the following: Perform an orient between the other point on the same line and one of the points on the other line and make note of which side the point on the other line is relative to the point on the same line. Do the same thing again with the other point on the other line. If the two points on the other line are on opposite sides from the point on the same line for all four points then the lines intersect.

b. For each line segment run the algorithm from part (a) on every other line segment and add that pair to an array if they intersect. This algorithm is brute force and takes $O(n^2)$ time which is much slower than the O(nlogn + k) algorithm.

c. This problem is NP-Complete, so it can only be done by brute force. Start by finding every combination of lines from the n lines. The number of combinations should come out to:

$$C(n,n) + C(n,n-1) + C(n,n-2) + \ldots + C(n,1)$$

Then, for each of these combinations of lines, determine if there are any intersections, and if there are, then ignore that combination. Out of the remaining combinations, whichever one has the most lines is the answer.

4)

```
def P(a,b):
    var arr[a][b]
    arr[0][0] = .5
    for i = 0; i < a; i++:
     for j = 0; j < b; j++:
       if a != 0 and b != 0:
             if a == 0:
                   arr[a][b] = .5 * arr[a][b-1]
             elif b == 0:
                   arr[a][b] = .5 * arr[a-1][b]
             else:
                   arr[a][b] = .5 * arr[a-1][b] + .5 * arr[a][b-1]
    return arr[a-1][b-1]
```

6) a. In order to find the shortest path to the right side of the cylinder you need to find the lengths of the shortest paths to each element in the last column. Each of these can be calculated as the minimum of the distances to the 3 elements in the previous column that can reach the current element, plus the current element. The base case is when you get down to the first column, and its distances are just the values of its elements. The recurrence equation for the recursive part of this algorithm is T(n) = 3T(n-1) which is $O(3^n)$, but the recursive

part needs to be performed n times, making the algorithm $O(n3^n)$ overall.

b. Start by creating a new nxn matrix and setting the values of the first column to the same as the values in the first column of the original matrix. Then go through the new matrix setting values so that

```
new[i][j] = old[i][j] + min(new[i-1][j-1], new[i][j-1], new[i+1][j-1])
```

This way, the algorithm is dynamic and building up from the bottom instead of recursing down and repeating calculations. The dynamic approach is O(n2) because it takes O(n2) time to fill in an nxn matrix.

7) a. Create a recursive function C(i,j) where *i* denotes the maximum currency denomination allowed (e.g. 1 for pennies, 2 for nickels, 3 for dimes) and *j* denotes the total number of cents to find combinations for. If j=0 then return 1, if j < 0 return 0, or if i=0 return 0. Otherwise return C(i-1,j) + C(i, j-i*) where i* is the i^{th} denomination.

b. Make a 2d array where the columns represent the denominations and the rows represent the value you are trying to get (a 5 X j array). Set the first row and column to 1. Loop through starting at [1, 1] always skipping the first column and for each box [i, j] insert the value of [i-1, j]+[i, j-i*].

8) a. Loop through all the options dividing the cost by the wait and tracking the maxumum. Fill the "backpack" with the item with the highest cost to weight ratio.

b. O(n) because you loop through all of the options once.

9) a. When $s_i = t_j$ the minimum of C[*i*-1*, j*]+1, C[*i, j*-1]+1, and C[*i*-1, *j*-1] represents the fact that a deletion, insertion, or replacement respectively is the most efficient way to get the search string. When s_i does not equal t_j C[*i*-1*, j*]+1, C[*i, j*-1]+1, and C[*i*-1, *j*-1]+1 represents the fact that a deletion, insertion, or replacement respectively is the most efficient way to get the search string.

b. C(0,j) = 0; no changes needed to the make substring an empty substring
C(i, 0) = i; There are i insertions needed.

c. Setup a 2d array of size mxp for each index, t_j in the text string, in which for each column, j, j[0] = 0, and for each row, i, i[0] = i. Start looping through starting at [1, 1] and for each index (skipping the first column) if t_j equals the i^{th} character in the search string, set that index in the array to the minimum of C[*i*-1*, j*]+1, C[*i, j*-1]+1, and C[*i*-1, *j*-1]. If tj does not equal the i^{th} character in the search string, set the index to C[*i*-1*, j*]+1, C[*i, j*-1]+1, and C[*i*-1, *j*-1]+1. The value in the very bottom right corner is the distance apart the search string and the text string are. Whichever index of the text string has the smallest distance from the search string is the most optimal j.

d. Starting at the very bottom right of the matrix made in c, find the minimum of C[*i*-1, *j*]+1, C[*i, j*-1]+1, and C[*i*-1, *j*-1]. If the minimum is C[*i*-1, *j*]+1 its a deletion, if the minimum is C[*i, j*-1]+1 it's an insertion, and if the minimum is C[*i*-1, *j*-1] it's a replace.