CC 401

1. To find the median in two separate databases in $O(\log n)$ time, we need to use the divide and conquer algorithm.

Algorithm: eliminate $k/2$ elements in each step.

```
def find_median(query1, query2, n):
    def find_kth(start1, start2, n):
        if start1 ≥ n:
            return query2(start2 + k-1)
        if start2 ≥ n:
            return query1(start1 + k-1)
        if k==1:
            return min(query1(start1), query2(start2))

        i = min(k//2, n-start1)
        j = k-i
        val1 = query1(start1 + i - 1)
        val2 = query2(start2 + j - 1)

        if val1 < val2:
            return find_kth(start1+i, start2, k-i)   # skip first i elements of db1
        else:
            return find_kth(start1, start2+j, k-j)   # skip first j elements of db2

    return find_kth(0,0,n)
```

Time: Since k=n, this runs in $O(\log n)$ time.

2a.) We can use dynamic programming to solve this question in O(U) complexity.
Treat dp[i] as the fewest number of coins to reach value i. Then, we will build the sub problems by dp[i] = min(dp[i], dp[i-d]+1)

Algorithm:

```
def solve(denom, U):
    dp = [INF] * (U+1)
    dp[0] = 0 # It takes 0 coins to reach 0.

    # Tabulation, build bottom-up.
    for i in range(1, U+1):
        for c in denom:
            if c ≤ i: # to avoid bound
                dp[i] = min(dp[i], dp[i-c]+1)
    return dp[U] # Returns INF if impossible
```

Explanation: For each value i, we try using each denomination and take the min number of coins needed.
The space complexity is O(U) because of the dp[] array.

2b.) To get the coins used, we need to do path reconstruction.

For coin[i], this represents the coin used up to this point.

Due to redudancy, I will extend the code from part 2a.)

```
# Add this array:
coin = [None] * (v+1)

# inside the denom loop:
  if c ≤ i and dp[i] ≥ dp[i-c]+1:
      coin[i] = c
      dp[i] = dp[i-c]+1

# Construct the path here: (outside)
used = []
while v > 0:
    used.append(coin[v])
    v = v - used[-1] # backtrack

print(used)
```

3.) To find an optimal plan, we can use dp where the state is (index, current_location).

To be more specific, say you are at month index in Chicago, then dp[index][Chicago] =
min(dp[index+1][STLouis] + M + C[index],
dp[index+1][Chicago] + C[index])

this approach is for memoization (top down)

Algorithm:

```
def solve(M, C, S):
    n = len(C)
    dp = [[INF]*2 for _ in range(n)]
        # col 0: chicago, col 1: st. Louis
    dp[0][0], dp[0][1] = 0,0  # cost is 0 in index 0.
    for i in range(1, n):
        dp[i][0] = min(dp[i-1][0] + C[i],
                       dp[i-1][1] + C[i] + M)
        dp[i][1] = min(dp[i-1] + S[i],
                       dp[i-1] + M + S[i])
    return min(dp[-1][0], dp[-1][1])
```

Time is linear O(n) because we only go through it once.

Space is linear O(n) as we use dp to save subproblems.

(4.) Analyzing the properties, we can safely assume that the graph is a DAG. So, there are no cycles and thus DP is appropriate.

* To find the longest path, we can just do:

as long as edge exists → $dp[i] = \min(dp[i], dp[j]+1)$ where $i < j$ for j in range$(i+1, n)$. We can use recursion and cache our result, or use tabulation:

```
def solve(edges, n):  # adjacency list
    dp = [0] * n
    for j in range(1, n):
        for i in range(j):  # i < j
            if (i, j) in graph:
                dp[j] = min(dp[j], dp[i]+1)
    return dp[-1]
```

* The bounds are flipped when doing the tabulation approach because dp[j] is built upon previous indices. Whereas, the initial explanation is fit for a top-down approach.

* The time complexity is $O(n + \#\text{ of edges})$.

5. To output the actual alignment, we need to construct our path through backtracking.

```
# Add another min array to store our
# path
D = [[0]*(n+1) for _ in range(m+1)]
        # 0 => from i-1, j-1
          1 => from i-1, j
          2 => from i, j-1

For j=1,...,n
    For i=1,...,m
        a = a_{x_i y_j} + A[i-1][j-1]
        b = δ + A[i-1][j]
        c = δ + A[i][j-1]
        A[i][j] = min(a, b, c)
        if A[i][j] == a:
            D[i][j] = 0
        elif A[i][j] == b:
            D[i][j] = 1
        else:
            D[i][j] = 2
# Path Reconstruction:
x, y = [], []
i, j = m, n
# start backtracking
while i>0 and j>0:
    if i-1 ≥ 0 and j-1 ≥ 0 and D[i][j] = 0:
        x.append(X[i-1])
        y.append(Y[j-1])
        i, j = i-1, j-1 # decrement
```

```
elif i-1≥0 and D[i][j]==1:
    x.append(X[i-1])
    y.append(None)
    i=i-1
elif j-1≥0 and D[i][j]==2:
    x.append(None)
    y.append(Y[j-1])
    y -=1
else:
    pass # should not get here
end of while
    print(x[::-1], y[::-1]) # output our path
Return A[M,N]
```