In [ ]:

```
def forwardStep(outcomes, a, b, priors):
    # Setting up Alpha Vector with Zeros
    alpha = np.zeros((outcomes.shape[0], a.shape[0]))
    alpha_norm = np.zeros((outcomes.shape[0], a.shape[0]))

    # Initializing Alpha vectors at starting point
    alpha[0] = priors * b[:,outcomes[0]]

    # Induction Step, updating alphas
    for t in range(outcomes.shape[0]-1):
        for i in range(a.shape[0]):
            alpha[t+1,i] = b[i,outcomes[t+1]] * alpha[t].dot(a[:,i])

    return alpha

def backwardStep(outcomes,a,b):
    # Initialize Betas
    beta = np.ones((outcomes.shape[0], a.shape[0]))

    # Backward Induction
    for t in range(outcomes.shape[0] -2,-1,-1):
        for i in range(a.shape[0]):
            beta[t,i] = (beta[t+1] * b[:,outcomes[t+1]]) @ a[i,:]

    return beta

def baumWelch(outcomes,a,b,priors, max_iters =100):
    T = len(outcomes)
    # Repated steps
    for r in range(max_iters):

        # Get Alphas and Betas at current step
        alpha = forwardStep(outcomes,a,b,priors)
        beta = backwardStep(outcomes,a,b)

        # Initializing xi matrix
        xi = np.zeros((a.shape[0],a.shape[0],T-1))

        # Filling XI up
        for t in range(T-1):
            # Denominator is always the same
            denominator = (alpha)[t,:] @ a * b[:, outcomes[t + 1]] @ beta[t + 1, :]
            for i in range(a.shape[0]):

                numerator = alpha[t, i] * a[i, :] * b[:, outcomes[t + 1]].T * beta[t + 1, :].T
                xi[i, :, t] = numerator / denominator

        # We define Gamma as the sum of the corresponding Xis, see tutorial by Rabiner
        gamma = np.sum(xi, axis = 1)

        # Updating Transition probabilities
        a = np.sum(xi, 2) / np.sum(gamma, axis=1).reshape((-1, 1))

        # Add additional T'th element in gamma
        gamma = np.hstack((gamma, np.sum(xi[:, :, T - 2], axis=0).reshape((-1, 1))))

        # Updating emission probabilities
        K = b.shape[1]
        denominator = np.sum(gamma, axis=1)
        for l in range(K):
            # Calculate each b for given state l
             b[:, l] = np.sum(gamma[:, outcomes == l], axis=1)

        # Divide by the common denominator
        b = np.divide(b, denominator.reshape((-1, 1)))

    # Return transistion and emission probabilities
    return a , b
```