```
#
# Edit Distance
#

#
# Dynamic programming solution: basic, recursive implementation
#
def edit_distance(A,B):
    return ED(A,B,0,0)

def ED(A, B, i, j):
    if i == len(A):
        return len(B) - j
    if j == len(B):
        return len(A) - i
    if A[i] == B[j]:
        c = 0
    else:
        c = 1
    return min(ED(A, B, i + 1, j + 1) + c, # change
               ED(A, B, i, j + 1) + 1,     # insert
               ED(A, B, i + 1, j) + 1)     # delete

#
# Efficient recursive implementation using "memoization"
#
def edit_distance_mem(A,B):
    return ED_mem(A,B,0,0,{})

def ED_mem(A,B,i,j,M):
    if (i,j) in M:               # If we have the solution in cache (M)
        return M[(i,j)]          # that is what we return. Otherwise
    if i == len(A):              # we compute the solution.
        return len(B) - j
    if j == len(B):
        return len(A) - i
    d = min(ED_mem(A, B, i + 1, j + 1, M) + (0 if A[i] == B[j] else 1), # change
            ED_mem(A, B, i, j + 1, M) + 1,                              # insert
            ED_mem(A, B, i + 1, j, M) + 1)                              # delete
    M[(i,j)] = d                 # we "memoize" the solution
    return d

#
# Efficient, iterative solution
#
def edit_distance_itr(A, B):
    n = len(A)
    m = len(B)
    D = [0]*(n+1)                # Distances at level j in B
    D_prev = [0]*(n+1)           # Distances at level j - 1 in B
    for i in range(n+1):
        D[i] = i

    for j in range(1, m+1):
        D, D_prev = D_prev, D
        D[0] = D_prev[0] + 1
        for i in range(1, n + 1):
            D[i] = min(D_prev[i] + 1,
                       D[i - 1] + 1,
                       D_prev[i - 1] + (0 if A[i-1] == B[j-1] else 1))
    return D[n]
```