Assignment 11b

A. Recursive Fibonacci

```
function fibonacci(n)
   if n == 0 or n == 1
      return n
   else
      return fibonacci(n-1) + fibonacci(n-2)
```

B. Memoized Fibonacci

```
function memoized_fibonacci(n, memo)
   if memo[n] is not null
      return memo[n]
   if n == 0 or n == 1
      memo[n] = n
      return n
   else
      memo[n] = memoized_fibonacci(n-1, memo) + memoized_fibonacci(n-2, memo)
      return memo[n]
```

C. Iterative Fibonacci

```
function iterative_fibonacci(n)
  if n == 0
    return 0
  if n == 1
    return 1

  fib_n_minus_2 = 0
  fib_n_minus_1 = 1
  fib_n = 0

  for i = 2 to n
    fib_n = fib_n_minus_1 + fib_n_minus_2
    fib_n_minus_2 = fib_n_minus_1
    fib_n_minus_1 = fib_n

  return fib_n
```

B. Longest Common Subsequence

Recursive LCS:

```
function lcs_recursive(X, Y, m, n)
  if m == 0 or n == 0
    return 0
  if X[m-1] == Y[n-1]
    return 1 + lcs_recursive(X, Y, m-1, n-1)
  else
    return max(lcs_recursive(X, Y, m, n-1), lcs_recursive(X, Y, m-1,[1] n))
```

Memoized LCS:

```
function lcs_memoized(X, Y, m, n, memo)
  if memo[m][n] != -1
    return memo[m][n]
  if m == 0 or n == 0
    memo[m][n] = 0
    return 0
  if X[m-1] == Y[n-1]
    memo[m][n] = 1 + lcs_memoized(X, Y, m-1, n-1, memo)
  else
    memo[m][n] = max(lcs_memoized(X, Y, m, n-1, memo), lcs_memoized(X, Y, m-1, n, memo))
  return memo[m][n]
```

- The recursive Fibonacci approach has exponential time complexity due to redundant calculations.
- The memoized Fibonacci approach uses a memoization table to store intermediate results, significantly reducing the number of recursive calls and improving efficiency.
- The iterative Fibonacci approach is the most efficient, with linear time complexity.
- The recursive LCS approach also suffers from exponential time complexity.
- The memoized LCS approach uses a 2D memoization table to store intermediate results, improving efficiency.