Iterative code can often be used to transform recursive code into a "nice" underlying structure. Recursive algorithms work best with optimization problems that have a technique for avoiding redundant work in Dynamic Programming.
Toy Example: Fibonacci

- $F(n)$:
  
  ```
  if n == 1 or n == 2 then return 1
  else return $F(n-1) + F(n-2)$
  ```

- What is the runtime of $F$? For a quick lower bound, change the last line to read:
  
  ```
  return $F(n-2) + F(n-2)$
  ```

Easy to see runtime is $\Omega(2^n)$, exponential!
Fixing Fibonacci, Part 1

• F(n):
  \[
  \text{static int a[MAX] \quad // initialized to all 0}
  \text{if a[n] \neq 0 then return a[n]}
  \text{if n == 1 or n == 2 then a[n] = 1}
  \text{else a[n] = F(n-1) + F(n-2)}
  \text{return a[n]}
  \]

• What is the new runtime of F?
  - One addition operation per array entry, so total runtime is $O(n)$. Linear!

• Still uses recursion. Can we do better?
  - How is the array actually filled in?
Fixing Fibonacci, Part 2

• F(n):
  
  ```
  int a[MAX]    // Need not be initialized
  for j = 3 to n
    a[j] = a[j-1] + a[j-2]
  return a[n]
  ```

• Recursion has been eliminated and replaced with a simple loop that builds the array directly
  • Recursive calls are replaced by table lookups

• Runtime remains \( O(n) \)
Fixing Fibonacci, Part 3

• $F(n)$:
  
  ```
  int a[MAX]  // Need not be initialized
  for j = 3 to n
    j1 = 1 + ( j mod 2 )
    j2 = 1 + ( j -1 mod 2)
    a[ j1 ] = a[ j1 ] + a[ j2 ]
  return a[ 1+ (n mod 2) ]
  ```

• Table has been compressed to constant space
• Runtime remains $O(n)$
Tackling Optimization Problems

• Instead of finding the optimal solution itself, try to find only the “cost” of the optimal solution
• Find a recursive program (using a minimal number of parameters) to solve for the cost
• Apply dynamic programming, building a table to avoid excess work (as in the Fibonacci example)
• Convert to iterative code if desired
• “Memoize” the table entries in order to recover the optimal solution itself
The LCS Problem

- Given seqs. \( X = \{x_1, \ldots, x_m\} \) and \( Y = \{y_1, \ldots, y_n\} \), find the longest common subsequence
  - A subsequence is formed by removing zero or more elements from a sequence
  - A common subseq. is a subseq. shared by \( X \) and \( Y \)
- Example: \( X = \{A,P,P,L,E,S\} \) \( Y = \{P,E,A,R,S\} \)
  - The LCS is \( \{P,E,S\} \)
  - May not always be unique (e.g. \( Y = \{P,E,A,R,L,S\} \))
- Naïve solution: Try all possible subsequences
  - What is the runtime? (Hint: Not good!)
Recursive sol’n for length of LCS

• $X_i$ denotes first $i$ elements of $X$, similarly, $Y_j$ ...

• LCSLen($X, Y$):
  
  $m = \text{len}(X); n = \text{len}(Y)$
  
  if $m == 0$ or $n == 0$ then return 0
  
  else if $x_m == y_n$ then return LCSLen($X_{m-1}, Y_{n-1}$) + 1
  
  else return max( LCSLen($X_{m-1}, Y_n$), LCSLen($X_m, Y_{n-1}$) )

• Easy to argue correctness

• Worst case runtime is exponential (analysis is similar to Fibonacci)
A Dynamic Programming Fix

• LCSLen(X,Y):
  static int c[MAXLEN,MAXLEN] // Initialized to all -1
  m = len(X); n = len(Y)
  if c[m,n] != -1 then return c[m,n]
  if m == 0 or n == 0 then c[m,n] = 0
  else if x_m == y_n then c[m,n] = LCSLen(X_{m-1},Y_{n-1}) + 1
  else c[m,n] = max( LCSLen(X_{m-1},Y_n), LCSLen(X_m,Y_{n-1}) )
  return c[m,n]

• Worst case runtime is now O(mn)
  ▪ Constant time per table entry, table will be m by n
• How is the table actually being built?
Iterative sol’n for length of LCS

• LCSLen(X,Y):
  static int c[MAXLEN,MAXLEN] // Initialized to all 0
  m = len(X); n = len(Y)
  for i = 1 to m
    for j = 1 to n
      if x_i == y_j then c[ i, j ] = c[ i-1, j-1] + 1
      else c[ i, j ] = max( c[ i-1, j ] , c[ i, j-1] )
    return c[m,n]

• Runtime is clearly still O(mn)
• No recursion
Finding the LCS: Memoize

- LCSTable(X,Y):
  static int c[MAXLEN,MAXLEN] // Initialized to all 0
  static int b[MAXLEN,MAXLEN] // May be uninitialized
  m = len(X); n = len(Y)
  for i = 1 to m; for j = 1 to n
    if x_i == y_j then
      c[i, j] = c[i-1, j-1] + 1; b[i, j] = 1 // Case 1
    else if c[i-1, j] > c[i, j-1] then
      c[i, j] = c[i-1, j]; b[i, j] = 2 // Case 2
    else
      c[i, j] = c[i, j-1]; b[i, j] = 3 // Case 3

- Need a second algorithm to recover the LCS from the table b[. , .] and print it out
Printing out the LCS

- PrintLCS(X, b, i, j):
  - if i == 0 or j == 0 then return
  - if b[i, j] == 1 then
    - PrintLCS(X, b, i-1, j-1)
    - print \( x_i \)
  - else if b[i, j] == 2 then PrintLCS(X, b, i-1, j)
  - else PrintLCS(X, b, i, j-1)

- Runtime is \( O(m + n) \), very efficient
- Notice that it was convenient to recover the LCS solution recursively
Matrix Multiplication

• Matrix multiplication basics:
  ▪ Matrix multiplication is *not* commutative
  ▪ An $p \times q$ matrix can only multiply with a $q \times r$ matrix
  ▪ Using the standard technique, it takes $pqr$ scalar multiplications to compute the product

• Matrix Chain Multiplication Problem: Given a chain of $n$ matrices $(A_1, \ldots, A_n)$ such that $A_i$ is a $p_{i-1} \times p_i$ matrix, *parenthesize* the product $A_1 A_2 \cdots A_n$ to minimize the number of scalar multiplications required to compute the product
  ▪ Ex: Is it faster to compute $A_1 (A_2 A_3)$ or $(A_1 A_2) A_3$?
Naïve Solution

• Try all possible parenthesizations and compute the cost of each to find the minimum…

• How many possible parenthesizations of n matrices are there? Call this \( P(n) \). Observe \( P(1) = 1 \).
  ▪ For \( n > 1 \), we have \( P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \)
  ▪ Can show \( P(n) = \Omega(2^n) \)!

• Not practical for large matrix chains (when it is most critical to find the best parenthesization)
Recursive sol’n for minimum cost of a matrix chain multiplication

- Let $m[i,j]$ be the minimum cost of computing the product of the chain $(A_i, \ldots, A_j)$ ($i < j$)
  - Observe that if $i = j$ then $m[i , j] = 0$
  - $m[i,j] = \min_{i \leq k < j} \{m[i,k] + m[k+1, j] + p_{i-1} p_k p_j\}$
    - In other words, we try parenthesizing the chain at each possible location $k$, and find the cost of the multiplication using that parenthesization

- Directly implementing this recursive algorithm requires exponential time...
Iterative sol’n for matrix chain order

Matrix-Chain-Order(p[1..n+1])
  Initialize array m[n,n] to \( \infty \)
  for i = 1 to n
    m[i, i] = 0
  for L = 2 to n  // L is chain length
    for i = 1 to n – L + 1
      j = i+L-1
      for k = i to j-1
        q = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j
        if q < m[i,j] then
          m[i,j] = q  ;  s[i,j] = k  // Memoization
  • Runtime is now \( O(n^3) \) using space \( O(n^2) \)
All Pairs Shortest Paths

• Given a graph with weighted edges, find the shortest path between every pair of nodes
  ▪ No negative weight cycles allowed (neg. edges OK)
• Naïve solution is to apply Single Source Shortest Path algorithm $|V|$ times
  ▪ Dijkstra doesn’t work with negative edge weights
  ▪ Bellman-Ford is $O(|V||E|)$ per source, yielding a worst case runtime of $O(|V|^4)$
• Can we do better using dynamic programming?
  ▪ Not obvious at all
Floyd-Warshall: Recursive APSP

- \( w[n,n] \) is edge weight matrix for \( n \) vertex graph
- Only vertices numbered \( k \) or less will be allowed to become internal vertices for shortest paths
- \( \text{FW}(w[n,n], k) \):
  
  ```
  int c[n,n]; int d[n,n]  // May be uninitialized
  if k == 0 then return w[ . , . ]
  d[ . , . ] = \text{FW}(w[ . , . ], k-1)  // Dynamic programming?
  for i = 1 to n; for j = 1 to n
    c[ i, j ] = \min( d[ i, j ] , d[ i, k ] + d[ k, j ] )
  return c[ . , . ]
  ```
- Can prove \( \text{FW}(w[n,n], n) \) will be sol’n to APSP cost
Removing Recursion from F-W

- **FW(w[n,n]):**
  
  ```
  static int d[n,n,n]    // May be uninitialized
  copy w[ , , ] into d[ , , 0] // This is the “base case”
  for k = 1 to n
      for i = 1 to n; for j = 1 to n
          d[ i,j, k ] = min( d[ i,j, k-1] , d[ i,k, k-1] + d[ k,j, k-1] )
  return d[ , , , n]
  ```

- **Once again, observe how the table is filled**
  - Entries at \( k^{th} \) level depend only on \( k-1^{st} \) level
  - Careful attention to detail reveals that it is safe to overwrite previous levels during the loop, like this…
Removing Recursion from F-W

• FW(w[n,n]):
  static int d[n,n]   // May be uninitialized
  copy w[ . , . ] into d[ . , . ]   // This is the “base case”
  for k = 1 to n
    for i = 1 to n; for j = 1 to n
      d[ i,j ] = min( d[ i,j ] , d[ i,k ] + d[ k,j ] )
  return d[ . , . ]

• Runtime is still O(n^3), space is now O(n^2)
• Now we need to “memoize” to recover the actual shortest paths
Memoized F-W

- FWTable(w[n,n]):
  static int d[n,n]; b[n,n]  // Initialize b[n,n] to all -1
  copy w[ . , . ] into d[ . , . ]  // This is the "base case"
  for i = 1 to n; for j = 1 to n; if w[ i,j ] < INF then b[ i,j ] = 0
  for k = 1 to n
    for i = 1 to n; for j = 1 to n
      if d[ i , j ] > d[ i , k ] + d[ k , j ] then
        d[ i , j ] = d[ i , k ] + d[ k , j ];  b[ i , j ] = k

- Can recover the actual paths from table b[ . , . ]
Printing out an F-W Path

• PrintFW(b[n,n], i, j):
  if b[i, j] = -1 then print “No path”
  else if b[i, j] = 0 then print (“i “,” j “)”
  else
    PrintFW(b[ , ], i, b[i, j])
    PrintFW(b[ , ], b[i, j], j)
  return

• Notice the use of recursion in path recovery