Dynamic Programming

What is Dynamic Programming?

- Technique for avoiding redundant work in recursive algorithms
- Works best with optimization problems that have a “nice” underlying structure
- Can often be used to transform recursive code into iterative code

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):
  ```
  static int a[MAX] // initialized to all 0
  if a[n] != 0 then return a[n]
  if n == 1 or n == 2 then a[n] = 1
  else a[n] = F(n-1) + F(n-2)
  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[j] = 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

Recursive sol’n for length of LCS

• $X_1, \ldots, X_i$ elements of $X$, similarly $Y_1, \ldots, Y_j$
• 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($X_{m-1}, Y$), LCSLen($X, Y_{n-1}$) )

• Easy to argue correctness
• Worst case runtime is exponential (analysis is similar to Fibonacci)

A Dynamic Programming Fix

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

• 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):
  \[
  \text{static int c[MLEN,MAXLEN] } // \text{Initialized to all 0}
  m = \text{len}(X); n = \text{len}(Y)
  \]
  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 $c[i,j] = \text{max}(c[i-1,j], c[i,j-1])$
  return $c[m,n]$

• Runtime is clearly still $O(mn)$
• No recursion

Finding the LCS: Memoize

• LCSTable(X,Y):
  \[
  \text{static int b[MLEN,MAXLEN] } // \text{Initialized to all 0}
  \]
  for $i = 1$ to $m$
    for $j = 1$ to $n$
      if $x_i = y_j$ then $b[i,j] = 1$
      else if $c[i-1,j] > c[i,j-1]$ then $b[i,j] = 1$
      else if $c[i,j] > c[i-1,j-1]$ then $b[i,j] = 1$
      else $b[i,j] = 0$
  if $b[m,n] = 0$ then print “No LCS”
  else $\text{print LCS}(X,Y)$

• Need a second algorithm to recover the LCS from the table $b[i,j]$ and print it out
Printing out the LCS

- PrintLCS(X, b, i, j):
  - if i == 0 or j == 0 then return
  - if bi == 1 then
    - PrintLCS(X, b, i-1, j-1)
    - print xi
  - else if bi == 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 \( p_i \times p_{i+1} \) matrix, parenthesize the product \( A_1A_2\cdots A_n \) to minimize the number of scalar multiplications required to compute the product
  - Ex: Is it faster to compute \( A_1(A_2A_3) \) or \( (A_1A_2)A_3 \)?

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 \( m[i,j] = 0 \)
  - \( m[i,j] = \min_{k<i} \{ m[i,k] + m[k+1,j] + p_i p_k p_j \} \)
  - 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 \( n \)
  for \( k = i \) to \( n \) \# L is chain length
    for \( i = 1 \) to \( n \)
      for \( j = i+L-1 \)
        \( q = m[i,k] + m[k+1,j] + p_i p_k p_j \)
        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 \( O(|V|^3) \)
- 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
- $FW(w, n, k)$:
  - If $k == 0$ then return $w[ , ]$
  - If $i == 1$ to $n$ for $j == 1$ to $n$
    - $d[i,j] = FW(w, ..., k-1) + d[i,k] + d[k,j]$
  - Can prove $FW(w, n, n)$ will be sol’n to APSP cost

Removing Recursion from F-W

- $FW(w, n, k)$:
  - If $k == 0$ then return $w[ , ]$
  - For $i == 1$ to $n$ for $j == 1$ to $n$
    - $d[i,j] = min(d[i,j], d[i,k] + d[k,j])$
  - 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

- $FWTable(w, n, k)$:
  - Initialize $b[n,n]$ to all -1
  - For $i == 1$ to $n$ for $j == 1$ to $n$
    - $d[i,j] = min(d[i,j], d[i,k] + d[k,j])$
  - Can recover the actual paths from table $b[ , ]$

Printing out an F-W Path

- $PrintFW(b[n,n], i, j)$:
  - If $b[i, j] == 0$ then print “No path”
  - Else if $b[i, j] > 0$ then print “(i, “ "$ j” ”
  - Else
    - $PrintFW(b[ , ], i, j)$
    - $PrintFW(b[ , ], b[i, j])$
  - Return

- Notice the use of recursion in path recovery