Dynamic Programming

Ming-Hwa Wang, Ph.D.
COEN 279/AMTH 377 Design and Analysis of Algorithms
Department of Computer Engineering
Santa Clara University

Dynamic Programming
- a bottom-up approach, rewrite the recursive algorithm as a non-recursive algorithm that systematically records the answers of the subproblems in a table
- overlapping subproblems and optimal substructure (the principle of optimality)
- variation: top-down memorization approach
- dynamic programming is most effective on well-ordered objects

Fibonacci Number
- $T(N) \geq T(N-1) + T(N-2)$, grows exponentially because to compute $F_{N,i}$ need 1 call to $F_{N-1,i}$, 2 calls to $F_{N-2,i}$, 3 calls to $F_{N-3,i}$, 5 calls to $F_{N-4,i}$, 8 calls to $F_{N-5,i}$, etc. $(3/2)^N \leq \text{Fib}(N) < (5/3)^N$
- record the two most recently computed Fibonacci numbers. $O(N)$

Binomial Coefficient
- the Pascal triangle
- $C(n,k) = \begin{cases} 
1 & \text{if } k = 0 \text{ or } k = n \\
C(n-1,k-1) + C(n-1,k) & \text{if } 0 < k < n \\
0 & \text{otherwise}
\end{cases}$

World Series Odds
- Two teams A and B are playing a match to see who is the first to win $n$ games. Suppose that A and B are equally competent, so each team has a 50% chance of winning any particular game.
- Let $P(i,j)$ be the probability that if A needs $i$ games to win, and B need $j$ games, that A will eventually win the match. $O(2^{i+j})$.
- $P(i,j) = \begin{cases} 
1 & \text{if } i = 0 \text{ and } j > 0 \\
0 & \text{if } i > 0 \text{ and } j = 0 \\
(P(i-1,j) + P(i,j-1))/2 & \text{if } i > 0 \text{ and } j > 0
\end{cases}$
- to avoid re-computing the same $P(i,j)$, just fill in a table, $O(n^2)$

Longest Increasing Sequence
- run has to be consecutive, but sequence’s selected elements need not be neighbors of each other
- define $li$ to be the length of the longest sequence ending with $s_i$: $O(n^2)$

Longest Common Subsequence (LCS)
- two sequence $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_m)$, the $i$th prefix of $X$ is $X_i = (x_1, x_2, ..., x_i)$. The length of LCS of $X$ and $Y$ is $|i,j|$ where:
  - $0$ if $i = 0$ or $j = 0$
  - $c[i-1, j-1] + 1$ if $i, j > 0$ and $x_i = y_j$
  - $\max(c[i, j-1], c[i-1, j])$ if $i, j > 0$ and $x_i \neq y_j$
- store $c[i, j]$ in row-major order, $b[i, j]$ points to the optimal subsolution. run time $O(mn)$, space $O(mn)$
- improve efficiency:
  - eliminate the $b$ table, which can be $O(1)$ computed
  - only need 2 rows of $c$ table at a time
- application: Unix diff

Ordering Matrix Multiplications
- matrix multiplication $A_1A_2...A_n$ is associative, but not commutative, $A_{pq} = C_{pr}$ with $pqr$ multiplications
- a fully parenthesizing of an expression corresponds to a parse tree.
- Catalan numbers $T(N) = \sum_{i=0}^{N-1}T(i)T(N-i) = C(2n-1, n-1)/n$ grows exponentially
- Let $c_i$ be the number of columns in matrix $A_i$ for $1 \leq i \leq N$. Then $A_i$ has $c_i$ rows and $C_0$ is the number of rows in $A_1$. Suppose $m_{Left, Right}$ is the number of multiplications required to multiply $A_{Left_1}A_{Left_2}...A_{Right_1}A_{Right_2}$ and $m_{Left, Left} = 0$. Suppose the last multiplication is $(A_{Left_1}...A_i)(A_{i+1}...A_{Right_2})$, where $Left_1 \leq i < Right_2$, then the number of multiplications used is $m_{Left,i} + m_{i+1,Right} + c_{Left-i}C_{Right}$. Let $M_{Left, Right}$ be the number of multiplications required in an optimal ordering, then $M_{Left, Right} = for Left_1 \leq i < Right_{min}(M_{Left,i} + m_{i+1,Right} + c_{Left-i}C_{Right})$
- run time $O(N^3)$, space $O(2^N)$

Optimal polygon triangulation
- A triangulation of a $n$-sided polygon corresponds to a parse tree with $n-1$ leaves. The internal nodes of the parse tree are the chords of the triangulation, and the root and leaves are the sides of the polygon. The triangle composed by the root and his two children split the polygon into two polygons and the triangle.
- Translate optimal polygon triangulation into ordering matrix multiplications: the $i$ sides become $c_i$. 
Optimal Binary Search Tree

- Given a list of words, \( w_1, w_2, \ldots, w_N \), and fixed probabilities \( p_1, p_2, \ldots, p_N \) of their occurrence. The problem is to arrange these words in a binary search tree in a way that minimizes the expected total access time. The number of comparisons needed to access an element at depth \( d \) is \( d+1 \), so if \( w_i \) is placed at depth \( d \), then we want to minimize \( \Sigma_{j=1}^i p_i (1+d) \).
- Using a greedy strategy, the word with the highest probability of being accessed was placed at the root. Not optimal.
- Using perfectly balanced binary tree, still not optimal.
- Optimal binary search trees do not constrain data to appear only at the leaves, also preserve the binary search tree property.
- Place the sorted words \( w_{left}, w_{left+1}, \ldots, w_{right}, w_{right} \) into a binary search tree. Suppose the optimal binary search tree has \( w_i \) (Left \( i \leq i \leq \) Right) as the root with cost \( C_{left, right} \), the left subtree must contain \( w_{left}, \ldots, w_{i-1} \) with cost \( C_{left,i-1} \), and the right subtree must contain \( w_{i+1}, \ldots, w_{right} \) with cost \( C_{i+1, right} \). Each node in these subtrees is one level deeper from \( w_i \) than from their respective roots, so we must add \( \Sigma_{j=left}^{i-1} p_j \) and \( \Sigma_{j=left+1}^{right} p_j \), this gives the formula:
  \[
  C_{left, right} = \text{for } Left \leq i \leq \text{Right} \min(p_i + C_{left,i-1} + C_{i+1, right} + \Sigma_{j=left}^{i-1} p_j + \Sigma_{j=left+1}^{right} p_j)
  \]

Linear partition Problem

- given an arrangement \( S \) of nonnegative numbers \( \{s_1, s_2, \ldots, s_n\} \) and an integer \( k \), partition \( S \) into \( k \) or fewer ranges so as to minimize the maximum sum over all the ranges
- let \( M[n,k] \) be the minimum possible cost over all partitioning of \( \{s_1, s_2, \ldots, s_n\} \) into \( k \) ranges, \( M[1,k] = s_1 \) for all \( k \geq 0 \), \( M[n,1] = \Sigma_i s_i \), \( M[n,k] = \min_{n=1}^{n} \max(M[i,k-1], \Sigma_{j=i}^{n} s_j) \), \( O(n^3) \)
- let \( n \) prefix sums \( p[i] = \Sigma_{k=1}^{i} s_k \), then \( \Sigma_{k=1}^{i} s_k = p[i] - p[i] \), \( O(kn^2) \)
- application: load balancing in parallel processing

Approximate String Matching

- let \( D[i,j] \) be the minimum number of differences between patterns \( P_1, P_2, \ldots, P_i \) and the segment of text \( T \) ending at \( j \).
  \[
  \begin{align*}
  & i \quad \text{if } j = 0 \\
  D[i,j] = & \begin{cases} 
  j & \text{if } i = 0 \\
  \min(D[i-1,j-1] + c(p_i, t_i), D[i-1,j] + 1, D[i,j-1] + 1) 
  \end{cases}
  \end{align*}
  \]
  Where \( c(p_i, t_j) \) is the match cost, \( c(p_i, t_j) = 0 \) if match, \( c(p_i, t_j) = 1 \) if substitute. \( D[i-1,j] \) is for insertion, and \( D[i,j-1] \) is for deletion. \( O(|P| \cdot |T|) \).
- application: animation