CS 241 – Analysis of Algorithms

Professor Eric Aaron

Lecture – T Th 9:00am

Lecture Meeting Location: OLB 205

Business

- HW5 extended, due November 19
  - HW6 to be out Nov. 14 or 15, due November 26

- HW4 grading update
  - Solution set 4 out today

- Reading: CLRS Ch. 11 (unstarred parts), Ch. 15.1-15.4
For the Most Part…

• The efficient Fibonacci methods used a characteristic technique of dynamic programming:
  – Results stored in a table (or similar), used to improve efficiency

• Dynamic programming solutions can be either top-down or bottom-up

• In general, when looking for a dynamic programming solution:
  – Try recursive, top-down approach with overlapping sub-problems
  – (Consider a memoized version)
  – Then, try bottom-up, iterative approach based on sub-problems
  – (Then, try to improve on space complexity of bottom-up method)

For the Most … Part 2

• Dynamic programming is often applied to optimization problems, to find a solution with an optimal (minimal or maximal) value
  – Often, for optimization problems, it is (or seems) necessary to consider all subsets of a set
  – … so, if we’re looking at a set of size \( n \), what’s the time complexity of such an algorithm?

• Characteristic structure for dynamic programming algorithms
  – Overlapping subproblems (as previously seen)
  – Optimal substructure: an optimal solution is built from the optimal solutions of subproblems

• Four parts in developing a dynamic programming algorithm:
  1. Characterize the structure of an optimal solution (in words)
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution from the bottom up
  4. Construct an optimal solution from computed information
Matrix Chain Multiplication

- Remember matrix multiplication (the standard way)?
  - If A is $i \times j$ and B is $j \times k$, $A \cdot B = C$ is $i \times k$, and the product runtime is $O(ijk)$
  - (Recall: matrix product is associative, not commutative)

- **Matrix chain multiplication** problem:
  - Given matrices $A_1, A_2, A_3, ..., A_n$, find the minimum number of multiplications needed to compute product $A_1 \cdot A_2 \cdot ... \cdot A_n$
    - That is, find the optimal way to parenthesize the matrix chain
  - Use dimension array $p$ s.t. for each $A_i$, its dimensions are $p_{i-1} \times p_i$

What is the range of indices in array $p$, for the above chain of $n$ matrices?

Dynamic Programming to the Rescue!

- Matrix chain multiplication is a candidate for a dynamic programming solution
  - Overlapping subproblems
  - Optimal substructure
    - How would we argue optimal substructure? How do we solve this problem so that an optimal solution is built on optimal solutions of subproblems?
      - "Suppose an optimal parenthesization of $A_iA_n\ldots A_j \ldots$"
- So, look for a dynamic programming solution
  - (What are the steps?)
    1. Characterize the structure of an optimal solution (in words)
    2. Recursively define the value of an optimal solution …
Matrix Chain Multiplication: A Recursive (Inefficient) Solution

- For the below, $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ is the chain of matrices to be multiplied.
- Array $p[0..n]$ stores the dimensions of each matrix.
  - Matrix $A_i$ is $p[i-1] \times p[i]$—or, in subscripts, $p_{i-1} \times p_i$.
- Then, an algorithm to compute the optimal number of multiplications for an ordering for this chain:

  \[ RCM(p, i, j) \]

  1. if $i == j$ then return 0 // no multiplication for a single matrix
  2. $M[i,j] = \infty$
  3. for $k = i$ to $j-1$
  4.     do $q = RCM(p, i, k) + RCM(p, k+1, j) + p_{i-1}p_kp_j$
  5.     if $q < M[i,j]$ then $M[i,j] = q$
  6. return $M[i,j]$

For the Most… Some More

- Recall the parts to developing a dyn. prog. algorithm:
  1. Characterize the structure of an optimal solution (in words)
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution from the bottom up
  4. Construct an optimal solution from computed information
- We have the first two. Now:
  - How to compute the value of an optimal solution (i.e., the number of computations) with bottom-up design instead of top-down recursion?
  - What additional information would support constructing a solution—an ordering for the chain multiplication—from the computation of the optimal value?

Stay with the same basic ideas, just expressed in a different design.
Bottom-up Matrix Chain Multiplication

`MC-Order(p)` // returns tables m, s

// let n = p.length-1; let m, s be new tables
for i = 1 to n
    m[i,i] = 0 // one matrix, no multiplying
for c = 2 to n // c is the chain length
    for i = 1 to n – c + 1
        j = i + c -1 // end index for chain of length c
        m[i,j] = \infty
        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]
                m[i,j] = q
                s[i,j] = k // the index for the optimal split

• What is the time complexity of this algorithm?
• What is the space complexity?
• What is the optimal value for the entire chain of n matrices?

And finally…
Finding a Solution from the Values

• That bottom-up method gives us the information from which we can get an optimal value and the associated indices
• How could we actually print the parenthesized / ordered chain of matrices?

`Print-Optimal-Parens (s, i, j)`
1. if i == j
2. then print "A",
3. else print "("
4. Print-Optimal-Parens(s, i, s[i,j])
5. Print-Optimal-Parens(s, s[i,j] +1, j)
6. print ")"