Dynamic Programming

1 Dynamic programming

• We have previously discussed how divide-and-conquer can often be used to obtain efficient algorithms.
  
  – Examples: matrix multiplication, merge sort, quicksort,....

• Sometimes direct use of divide-and-conquer does not yield efficient algorithms—in fact, sometimes it results in really bad algorithms.

• Today we will discuss a technique which can often be used to improve upon an inefficient divide-and-conquer algorithm.
  
  – The technique is called “Dynamic programming” for historical reasons. It really is neither especially “dynamic” nor especially “programming” related.
  
  – We will discuss dynamic programming by looking at an example.

1.1 Computing Fibonacci numbers

• The Fibonacci numbers $F_n$ are defined by the well-known recurrence

$$F_n = \begin{cases} 
F_{n-1} + F_{n-2} & \text{if } n \geq 2 \\
1 & \text{if } n = 1 \\
0 & \text{if } n = 0 
\end{cases}$$

The sequence for $n = 0, 1, 2, 3, \ldots$ is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ....

• The sequence is related to rabbits, how needles arrange themselves on pine cones, aesthetics, and graduate students, among other things.

• A naive recursive program to compute Fibonacci numbers based upon (1) would require $O(F_n)$ time, which is exponential in $n$!

• The reason is that the same subproblem is solved multiple times. For example, to compute $F(6)$, we need to compute $F(5)$ and $F(4)$. To compute $F(7)$, we recompute $F(6)$ and $F(5)$. The waste compounds recursively!
• The obvious remedy is to store the value of each Fibonacci number in a table when it is computed. Next time it is needed, we won’t have to recompute it from scratch, but rather we can just look up its value in the table.

\[ \Rightarrow O(n) \text{ time to compute } F_n \]!!

• As an aside, using generating functions or the trial-and-error method (see earlier handout), we can derive a closed-form expression for \( F_n \):

\[ F_n = \frac{1}{\sqrt{5}}(\Phi^n - \tilde{\Phi}^n), \] (2)

where \( \Phi = (1 + \sqrt{5})/2 \approx 1.61803 \ldots \) and \( \tilde{\Phi} = -1/\Phi \approx -0.61803 \) are the solutions to the equation \( 1 - z - z^2 = 0 \). We refer to \( \Phi \) as the golden ratio. Note that the second term in (2) is exponentially small. We can compute \( F_n \) simply by rounding \( \Phi^n/\sqrt{5} \) to the nearest integer.

• Therefore, using a table to store subproblem solutions reduces the running time from \( O(\Phi^n) \) to \( O(n) \). Pretty big improvement!

1.2 Matrix-chain multiplication

• Problem: Given a sequence of matrices \( A_1, A_2, A_3, \ldots, A_n \), find the best way (using the minimal number of multiplications) to compute their product.

  - Isn’t there only one way? \( ((\cdots ((A_1 \times A_2) \times A_3) \times \cdots) \times A_n) \)
  
  - No, matrix multiplication is associative;
    e.g., \( A_1 \times (A_2 \times (A_3 \times (\cdots \times (A_{n-1} \times A_n) \cdots))) \) yields the same matrix.

  - Different multiplication orders do not cost the same:
    
    * Multiplying \( p \times q \) matrix \( A \) and \( q \times r \) matrix \( B \) takes \( p \cdot q \cdot r \) multiplications; the result is a \( p \times r \) matrix.
    
    * Consider multiplying \( 10 \times 100 \) matrix \( A_1 \) with \( 100 \times 5 \) matrix \( A_2 \) and \( 5 \times 50 \) matrix \( A_3 \).
      
      - \( (A_1 \times A_2) \times A_3 \) takes \( 10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500 \) multiplications.
      
      - \( A_1 \times (A_2 \times A_3) \) takes \( 100 \cdot 5 \cdot 50 + 10 \cdot 50 \cdot 100 = 75000 \) multiplications.

• In general, let \( A_i \) be a \( p_{i-1} \times p_i \) matrix.

  - \( A_1, A_2, A_3, \ldots, A_n \) can be represented by the \( n + 1 \) integers \( p_0, p_1, p_2, p_3, \ldots, p_n \)

• Let \( m(i,j) \) denote the minimum number of multiplications needed to compute \( A_i \times A_{i+1} \times \cdots \times A_j \)

  - We want to compute \( m(1,n) \).

• Divide-and-conquer solution/recursive algorithm:

  - Divide the problem into \( j - i \) subproblems by considering the outer-level parentheses in all \( j - i \) possible positions. (E.g., \( (A_i \times A_{i+1} \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j) \) corresponds to multiplying a \( p_{k-1} \times p_k \) matrix by a \( p_k \times p_j \) matrix.)
- Recursively find best way of solving subproblems. (i.e., find the best way of computing \( A_i \times A_{i+1} \times \cdots \times A_k \) and the best way of computing \( A_{k+1} \times A_{k+2} \times \cdots \times A_j \))
- Pick best solution.

- Algorithm expressed in terms of \( m(i, j) \):
  \[
  m(i, j) = \begin{cases} 
  0 & \text{if } i = j \\
  \min_{i \leq k < j} \{m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j\} & \text{if } i < j
  \end{cases}
  \]

- Program:

  ```
  MATRIX-CHAIN(i, j)
  IF i = j THEN return 0
  m(i, j) = \infty
  FOR k = i TO j - 1 DO
    q = MATRIX-CHAIN(i, k) + MATRIX-CHAIN(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j
    IF q < m(i, j) THEN m(i, j) = q
  OD
  Return m(i, j)
  END MATRIX-CHAIN
  ```

  ```
  Return MATRIX-CHAIN(1, n)
  ```

- Running time: Let \( T(k) \) be the time to compute the optimal computation order for the product of \( k \) matrices. We define \( T(1) = 0 \).

  \[
  T(n) = \sum_{k=1}^{n-1} (T(k) + T(n - k) + c)
  = cn + 2 \sum_{k=1}^{n-1} T(k)
  = T(n - 1) + c + 2T(n - 1)
  = 3T(n - 1) + c
  \leq \frac{3c}{2} 3^{n-2}
  \]

- Easy to prove by induction or telescoping the recurrence.

- Problem is that, as in the Fibonacci example, we compute the same result over and over again.
  - Example: Recursion tree for \( \text{MATRIX-CHAIN}(1, 4) \)
For example, we compute \textsc{matrix-chain}(3, 4) twice.

- Solution is to “remember” values we have already computed in an \( n \times n \) table—\textit{memoization}

\begin{verbatim}
MATRIX-CHAIN(i, j)
  IF i = j THEN return 0
  IF m(i, j) < \( \infty \) THEN return m(i, j) /* This line has changed */
  FOR k = i to j - 1 DO
    q = MATRIX-CHAIN(i, k) + MATRIX-CHAIN(k + 1, j) + \( p_{i-1} \cdot p_k \cdot p_j \)
    IF q < m(i, j) THEN m(i, j) = q
  OD
  return m(i, j)
END MATRIX-CHAIN

FOR i = 1 to n DO
  FOR j = i to n DO
    m(i, j) = \( \infty \)
  OD
OD

return MATRIX-CHAIN(1, n)
\end{verbatim}

- Running time:
  - \( \Theta(n^2) \) different calls to \textsc{matrix-chain}(i, j).
  - The first time a call is made it takes \( O(n) \) time, \textit{not} counting recursive calls.
  - When a call has been made once it costs \( O(1) \) time to make it again.
    \( \implies O(n^3) \) time
  - Another way of thinking about it: \( \Theta(n^2) \) total entries to fill; it takes \( O(n) \) time to fill each one.
1.3 Alternative view of Dynamic Programming

- Often (including in the book) dynamic programming is presented in a different way, namely, as filling up a table from the bottom to the top.

- Matrix-chain example:
  Key is that $m(i, j)$ only depends upon $m(i, k)$ and $m(k + 1, j)$ where $i \leq k < j$.

  $\implies$ If we have already computed $m(i, k)$ and $m(k + 1, j)$ for each $k$, then we can compute $m(i, j)$ in $O(|j - i|)$ time by taking the appropriate minimum over all $k$.

  - We can easily compute $m(i, i)$ for all $1 \leq i \leq n$: $m(i, i) = 0$
  - Then we can easily compute $m(i, i + 1)$ for all $1 \leq i \leq n - 1$:
    $m(i, i + 1) = m(i, i) + m(i + 1, i + 1) + p_{i-1} \cdot p_i \cdot p_{i+1}$
  - Then we can compute $m(i, i + 2)$ for all $1 \leq i \leq n - 2$:
    $m(i, i + 2) = \min \{m(i, i) + m(i + 1, i + 2) + p_{i-1} \cdot p_i \cdot p_{i+2}, m(i, i + 1) + m(i + 2, i + 2) + p_{i-1} \cdot p_{i+1} \cdot p_{i+2}\}$
    $\vdots$
  - Until finally we compute $m(1, n)$ and we’re done.
  - Computation order is indicated by the numbers in the slots below:


- Program:
FOR $i = 1$ to $n$ DO  
  \[ m(i, i) = 0 \]
OD  
FOR $l = 1$ to $n - 1$ DO  
  FOR $i = 1$ to $n - l$ DO  
    $j = i + l$  
    $m(i, j) = \infty$  
    FOR $k = 1$ to $j - 1$ DO  
      $q = m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j$  
      IF $q < m(i, j)$ THEN $m(i, j) = q$  
    OD  
  OD  
OD

- Analysis:  
  - $O(n^2)$ entries, $O(n)$ time to compute each $\implies O(n^3)$ total time.

- Note:  
  - I like recursive (divide-and-conquer) thinking, because you don’t need a new idea (and write a totally new program)—just use table lookup!
  - Book seems to like bottom-up method better.

1.4 Overview of dynamic programming

Dynamic programming is a way of improving on inefficient divide-and-conquer algorithms. By “inefficient”, we mean that the same recursive call is made over and over.

- If same subproblem is solved several times, we can use table to store result of a subproblem the first time it is computed and thus never have to recompute it again.
- Alternatively, we can think about filling up a table of subproblem solutions from the bottom up.