Algorithms and Data Structures (CSci 115)

California State University Fresno

College of Science and Mathematics

Department of Computer Science

H. Cecotti

Learning outcomes

- **Dynamic programming**
  - Definitions and principles
  - Examples

Introduction

- We have done dynamic programming before calling it dynamic programming…
  - It is a technique, not an algorithm
- Dynamic programming (DP)
  - Solving problems by **combining** the solution to sub-problems
- Remark
  - **Divide & Conquer**: partition the problem into **disjoint** sub-problems
  - **Dynamic programming**: the sub-problems **overlap**
    - Solves each sub-sub-problem one time and then saves its answer in a table
    - → Avoiding the work of recomputing the answer every time it solves each sub-sub-problem!
- Typical application
  - Optimization problem
    - Minimize/maximize a cost function

Dynamic programming

- Definitions
  - DP solves problems by combining solutions to sub-problems.
  - Principle:
    - Sub-problems are **not** independent.
    - Sub-problems **may share** sub-sub-problems,
    - Yet, a solution to one sub-problem may not affect the solutions to other sub-problems of the **same** problem.
  - DP reduces computation by:
    - Solving sub-problems in a bottom-up fashion.
    - Storing solution to a sub-problem the first time it is solved.
    - Looking up the solution when sub-problem is encountered again.
      - → **Table**

Dynamic programming

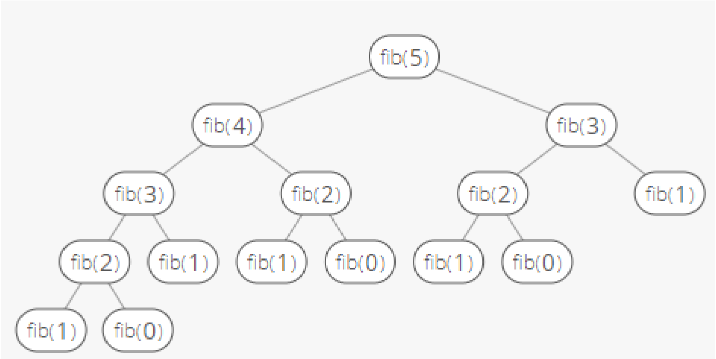
- **Steps**
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution
     - Bottom-up with a table
     - Top-down with caching
  4. Construct an optimal solution from computed information.

Memoization

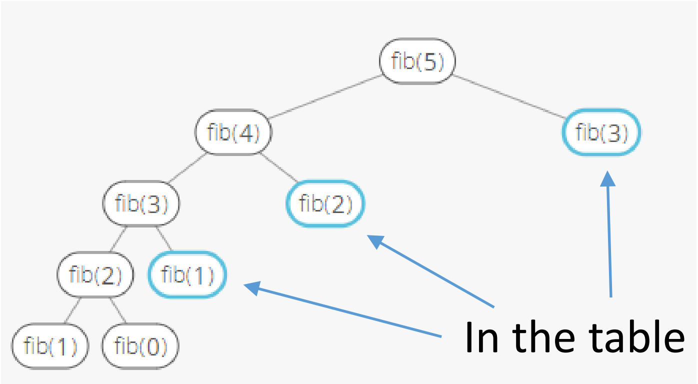
- Definition
  - **Memoization**
    - An optimization technique used primarily to speed up computer programs
      - By storing the results of **expensive** function calls
      - By returning the cached result when the same inputs occur again
    - It ensures that: a method doesn't run for the same inputs more than once by keeping a record of the results for the given inputs
    - Common strategy for dynamic programming problems
- Example
  - Remember the factorial function (recursive definition)
    - If it is computed already for a particular n, you don’t have to call it again and go deep until 0

Memoization

- Example with Fibonacci sequence
  - What will happen in the default case:
    - Recursive calls:



  - int result = fib(n - 1) + fib(n - 2);
    - Compute fib(n-1) then fib(n-2) ...
    - Check if it has been computed already !
      - For each new result, put it in a table, so when you dig in fib(n-2)
      - We just pick the result from the table



Matrix chain multiplication

- Goal:
  - ➢ Algorithm that solves the problem of matrix-chain multiplication
    - Input: a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied
    - Output: product $A_1A_2\ldots A_n$
- Remark
  - ➢ Using the standard algorithm for multiplying pairs of matrices
    - (subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together).
    - Matrix multiplication: **associative** → all parenthesizations give the same output
    - A product of matrices is **fully parenthesized** if
      - it is a single matrix or the product of 2 fully parenthesized matrix products - surrounded by parentheses
  - ➢ Example,
    - if $\langle A_1A_2A_3A_4 \rangle$ as input then 5 possibilities
      1. $(A_1(A_2(A_3A_4)))$
      2. $(A_1((A_2A_3)A_4))$
      3. $((A_1A_2)(A_3A_4))$
      4. $((A_1(A_2A_3))A_4))$
      5. $(((A_1A_2)A_3)A_4)$

Matrix chain multiplication

- Pseudo-code: Matrix Multiplication
  - Input: A (p x q) , B (q x r) → Output: C (p x r)
  - Computation: related to p*q*r
- Example:
  - We want to do $A_1A_2A_3$ → 2 possibilities
    - $(A_1A_2)A_3$ or $A_1(A_2A_3)$
    - A_1 = 10 x 100
    - A_2 = 100 x 5
    - A_3 = 5 x 50
  - Case 1: $(A_1A_2)A_3$
    - 10*100*5 = 5000 operations, output 10 x 5
    - 10*5*50 = 2500 operations, output 10 x 50
    - Total = 7500 operations
  - Case 2: $A_1(A_2A_3)$
    - 100*5*50= 25000 operations, output 100 x 50
    - 10*100*50 = 50000 operations, output 10 x 50
    - Total = 75000 operations
- → Computing the product in Case 1: 10x faster!

```
MATRIX-MULTIPLY(A, B)
if A.columns ≠ B.rows
    error "incompatible dimensions"
else let C be a new A.rows × B.columns matrix
    for i = 1 to A.rows
        for j = 1 to B.columns
            c_ij = 0
            for k = 1 to A.columns
                c_ij = c_ij + a_ik · b_kj
    return C
```

Matrix chain multiplication

- Problem definition
  - $<A_1,A_2,A_i\ldots A_n>$ sequence of n matrices $1 \le i \le n$ and A_i of size $p_{i-1} \times p_i$
- Goal
  - Only to find an order for multiplying matrices at the lowest cost
  - Remarks:
    - Time invested to find the order << time saved when performing the matrix multiplications
    - Exhaustively search finding all possible parenthesizations:
      - Not an efficient algorithm
  - Number of alternative parenthesizations of a sequence of n matrices:
    - P(n) such that:

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \ge 2 . \end{cases}$$

Matrix chain multiplication

- **Part 1**: Structure of an optimal parenthesization
  - Find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems
  - We consider: $A_{i..j}=A_i * \dots * A_j$, $i \le j$
  - Cost of $A_{i..j}$ = Cost of $A_{i..k}$ + Cost of $A_{k..j}$ + Cost multiplication $A_{i..k} * A_{k..j}$
- Now, suppose that to optimally parenthesize $A_{i..j}$ split the product between A_k and A_{k+1}
- Then
  - how we parenthesize the "prefix" subchain $A_{i..k}$ within this optimal parenthesization of $A_{i..j}$ must be an **optimal** parenthesization of $A_{i..k}$
- **Because**:
  - **If** there were a less costly way to parenthesize $A_{i..k}$
  - **Then** we can substitute that parenthesization in the optimal parenthesization of $A_{i..j}$ to produce another way to parenthesize $A_{i..j}$ whose cost was **lower** than the optimum: **a contradiction**!!
  - Same observation is true for how we parenthesize the subchain $A_{k+1..j}$ in the optimal parenthesization of $A_{i..j}$:
  - → must be an optimal parenthesization of $A_{k+1..j}$

Matrix chain multiplication

- **Task**:
  1. Split the problem into 2 sub-problems (optimally parenthesizing $A_{i..k}$ and $A_{k+1..j}$)
  2. Finding optimal solutions to sub-problem instances
  3. Combining these optimal sub-problem solutions.

Matrix chain multiplication

- **Part 2**: Recursive solution
  - Cost of an optimal solution recursively in terms of the optimal solutions to subproblems
  - Our sub-problems:
    - The problems of determining the minimum cost of parenthesizing $A_{i..j}$
    - Let m[i,j] : the minimum number of scalar multiplications needed to compute $A_{i..j}$
      - For the full problem, we have the lowest cost way to compute $A_{i..j}$ is m[1,n]
    - Definition
      - m[i,i]=0 for 1≤i≤n
      - $m[i,j]=m[i,k]+m[k+1,j] + p_{i-1}*p_k*p_j$
    - What we need: **s[i,j] to be a value of k at which we split**
  - Minimum cost definition

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \ . \end{cases}$$

Matrix chain multiplication

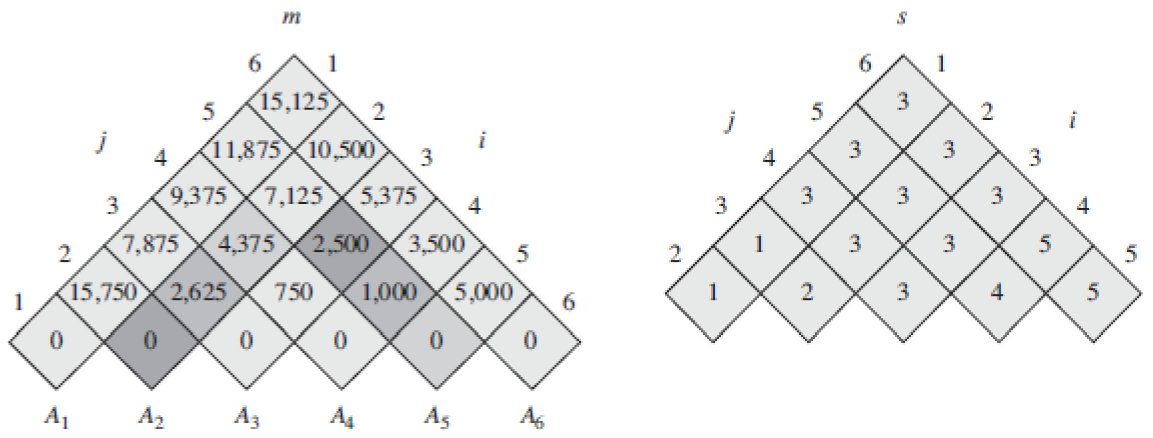
- **Part 3:** Computing the optimal costs
  - Instead of computing the solution to recurrence recursively
  - → we compute the optimal cost by using a tabular, bottom-up approach
  - The procedure uses an auxiliary table m[1..n,1..n]
    - for storing the m[i,j] costs and
    - Another auxiliary table s[1..n-1,2..n] that records which index of k achieved the optimal cost in computing m[i,j]
    - → use the table s to construct an optimal solution

Matrix chain multiplication

- Algorithm
  - ➢ Computes the rows from bottom to top and from left to right within each row

MATRIX-CHAIN-ORDER(p)

```
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n                                    Init
    m[i, i] = 0
for l = 2 to n              // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```



matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

Matrix chain multiplication

- **Part 4**: Constructing an optimal solution
  - Table s[1..n-1,2..n] gives us the solution

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

Conclusion

- Dynamic programming
  - **Key concept in programming**
    - For optimization problems
    - Very useful for a large number of problems
- Famous dynamic programming algorithms.
  - Viterbi algorithm for hidden Markov models
    - → generative model in machine learning
  - Unix diff for comparing 2 files.
  - Smith-Waterman for sequence alignment.
  - Bellman-Ford for shortest path routing in networks
  - Cocke-Kasami-Younger for parsing context free grammars.

Questions ?

- Reading
  - ➢Canvas: Csci 115 book - Section 9.5 + Section 10.3
  - ➢Recommended book chapter:
    - o Introduction to Algorithms 3th Edition, Chapter 15.