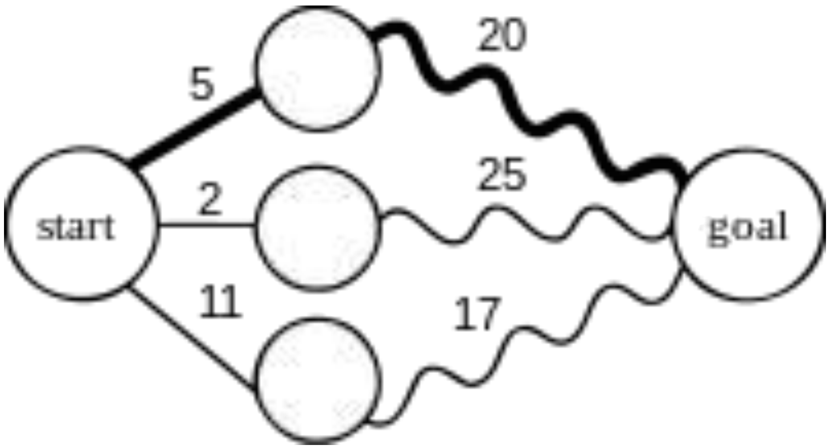


Dynamic Programming

Matrix Chain Multiplication

Dynamic Programming

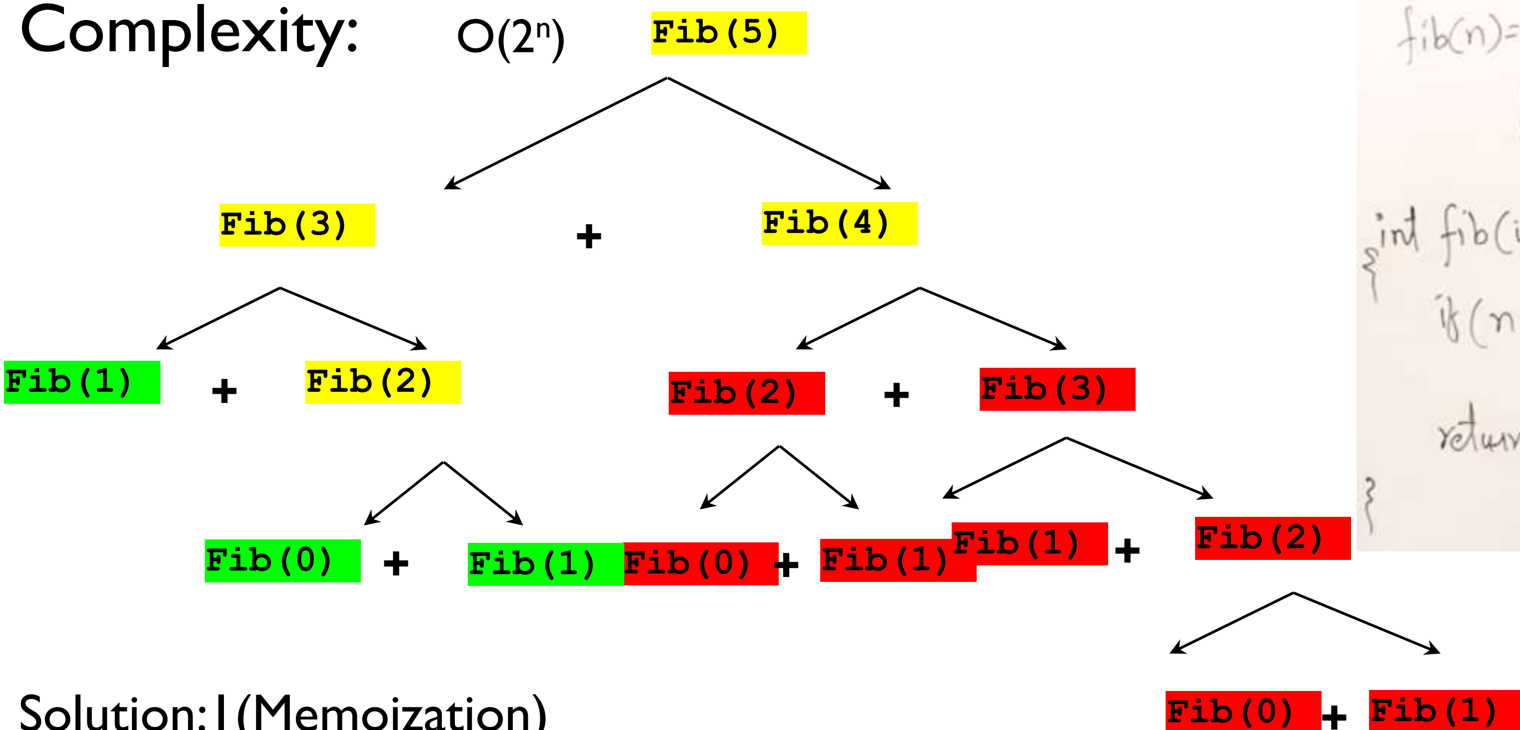
- Introduction
- Drawback of Recursion
- Elements of Dynamic Programming
- Matrix Chain Multiplication

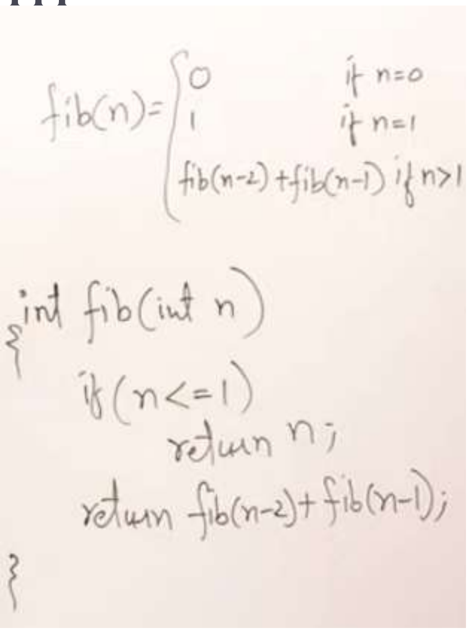
Dynamic Programming

- Like divide and conquer, DP solves problems by combining solutions from subproblems.
- Unlike divide and conquer, subproblems are not independent.
- DP reduces computation by
  - Solving subproblems in a bottom-up fashion.
  - Storing solution to a subproblem the first time it is solved.
  - Looking up the solution when subproblem is encountered again.
- Examples
  - Matrix Multiplication
  - Longest Common Subsequence

Drawbacks of Recursion: Recursion Tree for n^{th} Fibonacci Term

Complexity: $O(2^n)$





Solution: 1 (Memoization)

Complexity: $O(n)$

Top Down Approach

Fib(0)	Fib(1)	Fib(2)	Fib(3)	Fib(4)	Fib(5)
0	1	1	2	3	5

- Soultion2:
  - Dynamic Programming Approach
    - Tabulation Method
    - Bottom-up Approach

$$fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ fib(n-2)+fib(n-1) & \text{if } n>1 \end{cases}$$

```
int fib(int n)
{
    if(n<=1)
        return n;
    F[0]=0; F[1]=1;
    for(int i=2;i<=n;i++)
    {
        F[i]=F[i-2]+F[i-1];
    }
    return F[n];
}
```

F (0)	F(1)	F(2)	F(3)	F(4)	F(5)
0	1	1	2	3	5

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution in a bottom-up fashion
4. Construct an optimal solution from computed values.

Matrix Chain Multiplication

- Given : a chain of matrices $\{A_1, A_2, \ldots, A_n\}$.
- Once all pairs of matrices are *parenthesized*, they can be multiplied by using the standard algorithm as a sub-routine.
- A product of matrices is ***fully parenthesized*** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. [Note: since matrix multiplication is associative, all parenthesizations yield the same product.]

Matrix-chain Multiplication ...contd

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:

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. $(((A_1A_2)A_3)A_4)$

The way the chain is parenthesized can have a dramatic impact on the cost of evaluating the product.

Matrix-chain Multiplication ...contd

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 =2{,}500$ scalar multiplications

    Total: 7,500
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 =50{,}000$ scalar multiplications

    Total: 75,000

Matrix-chain Multiplication ...contd

- Matrix-chain multiplication problem
  - Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for $i=1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$
  - Parenthesize the product $A_1A_2\ldots A_n$ such that the total number of scalar multiplications is minimized
- Brute force method of exhaustive search takes time exponential in n
- eg:-

  $A_1(5x4), A_2(4x6), A_3(6x2)_), A_4\ (2x7)$

  $A_1(p_0xp_1), A_2(p_1xp_2), A_3(p_2xp_3)_), A_4\ (p_3xp_4)$

Dynamic Programming Approach

- The structure of an optimal solution
  - Let us use the notation $A_{i..j}$ for the matrix that results from the product $A_i A_{i+1} \dots A_j$
  - An optimal parenthesization of the product $A_1 A_2 \dots A_n$ splits the product between A_k and A_{k+1} for some integer k where $1 \le k < n$
  - First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$

$A_1(5x4), A_2(4x6), A_3(6x2)_), A_4\ (2x7)$

$A_1(p_0xp_1), A_2(p_1xp_2), A_3(p_2xp_3)_), A_4\ (p_3xp_4)$

Dynamic Programming Approach

…contd

- Recursive definition of the value of an optimal solution
  - Let $m[i, j]$ be the minimum number of scalar multiplications necessary to compute $A_{i..j}$
  - Minimum cost to compute $A_{1..n}$ is $m[1, n]$
  - Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1} for some integer k where $i \leq k < j$

Dynamic Programming Approach

…contd

- $A_{i..j} = (A_i A_{i+1} \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_j) = A_{i..k} \cdot A_{k+1..j}$
- Cost of computing $A_{i..j}$ = cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying $A_{i..k}$ and $A_{k+1..j}$
- Cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ is $p_{i-1} p_k p_j$

- $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \quad$ *for* $i \leq k < j$
- $m[i, i] = 0$ for $i = 1, 2, \ldots, n$

$A_1 (5x4), A_2 (4x6), A_3 (6x2)), A_4 (2x7)$

$A_1 (p_0 x p_1), A_2 (p_1 x p_2), A_3 (p_2 x p_3)), A_4 (p_3 x p_4)$

Dynamic Programming Approach

...contd

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

Dynamic Programming Approach

…contd

- To keep track of how to construct an optimal solution, we use a table *s*
- $s[i, j]$ = value of k at which $A_i A_{i+1} \ldots A_j$ is split for optimal parenthesization
- Algorithm:
  - First computes costs for chains of length l=1
  - Then for chains of length l=2,3, … and so on
  - Computes the optimal cost bottom-up

$A_1(5x4), A_2(4x6), A_3(6x2)_), A_4\ (2x7)$

$A_1(p_0 x p_1), A_2(p_1 x p_2), A_3(p_2 x p_3)_), A_4\ (p_3 x p_4)$

0			
	0		
		0	
			0

Chain of length 1

M[1,1]=M[2,2]=M[3,3]=M[4,4]=0

0	**120**		
	0	**48**	
		0	**84**
			0

Chain of length 2

M[1,2]=5x4x6=120

M[2,3]=4x6x2=48

M[3,4]=6x2x7=84

Split Matrix S

	1		
		2	
			3

$A_1(5x4), A_2(4x6), A_3(6x2)), A_4(2x7)$

$A_1(p_0 x p_1), A_2(p_1 x p_2), A_3(p_2 x p_3)), A_4(p_3 x p_4)$

Chain of length 3

M[1,3]

Case:1

M[1,1]+M[2,3]
=0+48+(5x4x2)
=**88**

Case:2

M[1,2]+M[3,3]
=120+0+(5x6x2)
=180

M[2,4]

Case:1

M[2,2]+M[3,4]
=0+84+(4x6x7)
=252

Case:2

M[2,3]+M[4,4]
=48+0+(4x2x7)
=**104**

0	120		
	0	48	
		0	84
			0

0	120	**88**	
	0	48	**104**
		0	84
			0

	1	**1**	
		2	**3**
			3

Chain of length 4

M[1,4]

Case:1

M[1,1]+M[2,4]
=0+104+(5x4x7)
=244

Case:2

M[1,2]+M[3,4]
=120+84+(5x6x7)
=414

Case:3

M[1,3]+M[4,4]
=88+0+(5x2x7)
=**158**

0	120	88	**158**
	0	48	104
		0	84
			0

	1	**1**	**3**
		2	**3**
			3

Algorithm to Compute Optimal Cost

Input: Array $p[0 \ldots n]$ containing matrix dimensions and n

Result: Minimum-cost table m and split table s

```
MATRIX-CHAIN-ORDER(p[ ], n)
    for i ← 1 to n
        m[i, i] ← 0
    for l ← 2 to n
        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
```

Takes $O(n^3)$ time

Requires $O(n^2)$ space

Constructing Optimal Solution

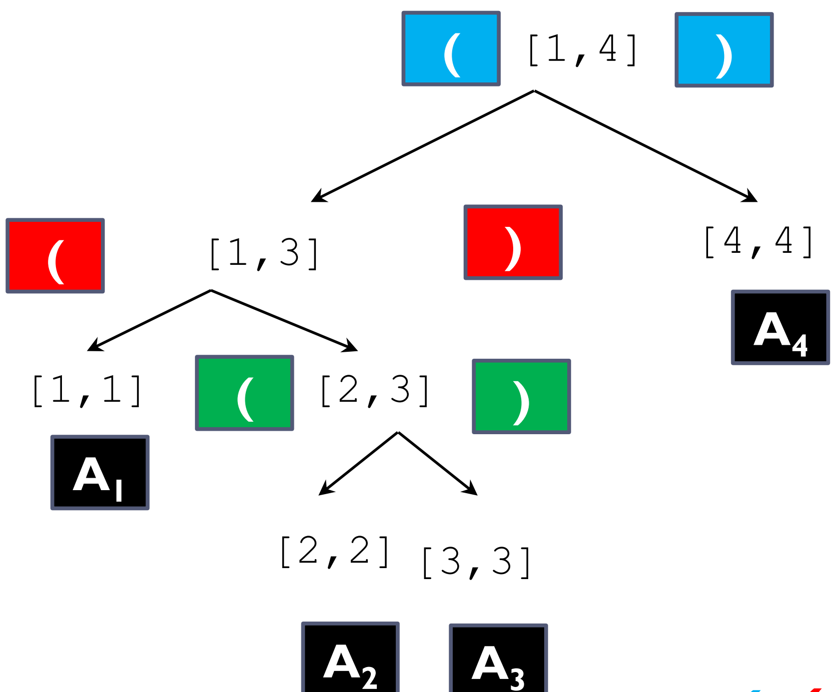
- Our algorithm computes the minimum-cost table *m* and the split table *s*
- The optimal solution can be constructed from the split table *s*
  - Each entry $s[i, j] = k$ shows where to split the product $A_i A_{i+1} \ldots A_j$ for the minimum cost

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

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

Split Matrix S

	1	1	3
		2	3
			3



$$((A_1 (A_2 A_3)) A_4)$$