Algorithmic Paradigms

Divide and Conquer

Idea: Divide problem instance into smaller sub-instances of the same problem, solve these recursively, and then put solutions together to a solution of the given instance.

Examples: Mergesort, Quicksort, Strassen's algorithm, FFT.

Greedy Algorithms

Idea: Find solution by always making the choice that looks optimal at the moment — don't look ahead, never go back.

Examples: Prim's algorithm, Kruskal's algorithm.

Dynamic Programming

Idea: Turn recursion upside down.

Example: Floyd-Warshall algorithm for the all pairs shortest path problem.

Dynamic Programming - A Toy Example

Fibonacci Numbers

$$
\begin{aligned}
F_0 &= 0, \\
F_1 &= 1, \\
F_n &= F_{n-1} + F_{n-2} \qquad \text{(for } n \geq 2\text{).}
\end{aligned}
$$

A recursive algorithm

Algorithm REC-FIB(n)

1. **if** $n = 0$ **then**
2. **return** 0
3. **else if** $n = 1$ **then**
4. **return** 1
5. **else**
6. **return** REC-FIB$(n - 1)$ + REC-FIB$(n - 2)$

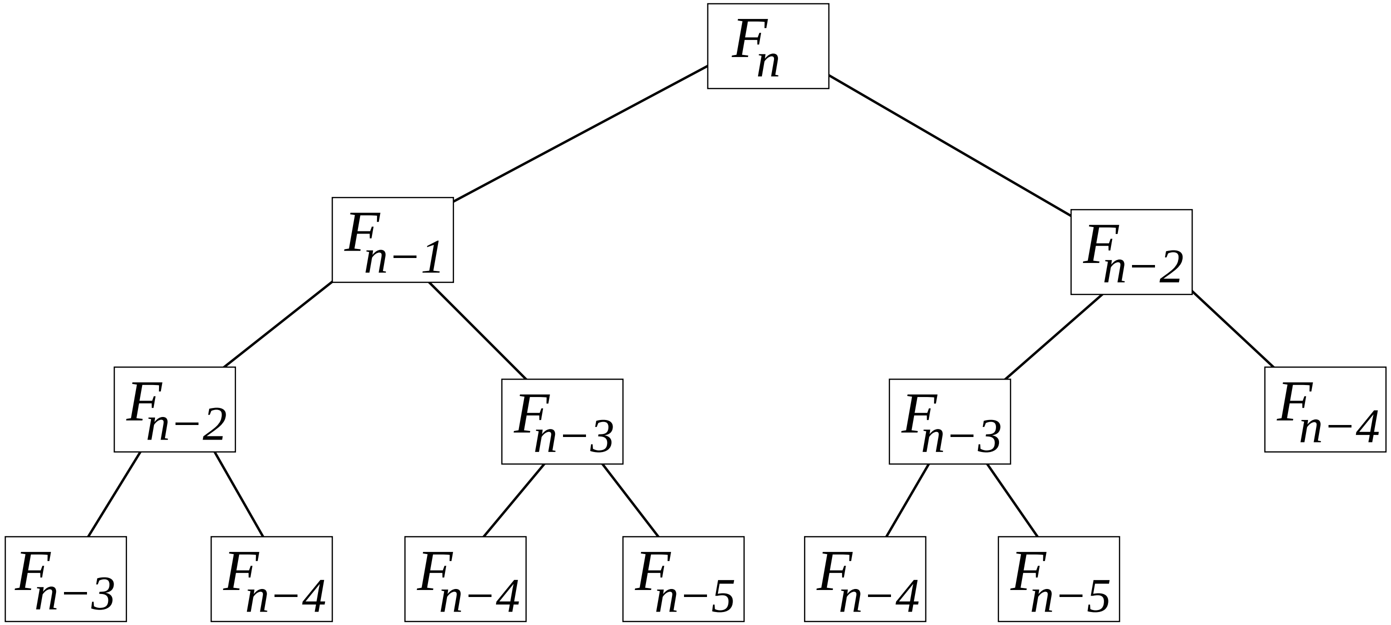
Ridiculously slow: **exponentially many** repeated computations of REC-FIB(j) for small values of j.

Fibonacci Example (cont'd)

Why is the recursive solution so slow?

Running time $T(n)$ satisfies

$$T(n) = T(n-1) + T(n-2) + \Theta(1) \geq F_n \approx 1.6^n.$$



Fibonacci Example (cont'd)

Dynamic Programming Approach

Algorithm DYN-FIB(n)

1. $F[0] = 0$
2. $F[1] = 1$
3. **for** $i \leftarrow 2$ **to** n **do**
4. $\quad F[i] \leftarrow F[i-1] + F[i-2]$
5. **return** $F[n]$

Running Time

$$\Theta(n)$$

Very fast in practice - just need an array (of linear size) to store the $F(i)$ values.

Multiplying Sequences of Matrices

Recall

Multiplying a $(p \times q)$ matrix with a $(q \times r)$ matrix (in the standard way) requires

$$pqr$$

multiplications.

We want to compute products of the form

$$A_1 \cdot A_2 \cdots A_n.$$

How do we set the parentheses?

Example

Compute

$$\begin{array}{ccccccc} A & \cdot & B & \cdot & C & \cdot & D \\ 30 \times 1 & & 1 \times 40 & & 40 \times 10 & & 10 \times 25 \end{array}$$

Multiplication order $(A \cdot B) \cdot (C \cdot D)$ requires

$$30 \cdot 1 \cdot 40 + 40 \cdot 10 \cdot 25 + 30 \cdot 40 \cdot 25 = 41,200$$

multiplications.

Multiplication order $A \cdot ((B \cdot C) \cdot D)$ requires

$$1 \cdot 40 \cdot 10 + 1 \cdot 10 \cdot 25 + 30 \cdot 1 \cdot 25 = 1,400$$

multiplications.

The Matrix Chain Multiplication Problem

Input:	Sequence of matrices $A_1, \ldots, A_n$,
	where A_i is a $p_{i-1} \times p_i$-matrix
Output:	Optimal number of multiplications needed to compute
	$A_1 \cdot A_2 \cdots A_n$
	and optimal parenthesisation

Running time of algorithms will be measured in terms of n.

Solution Attempts

Approach 1: Exhaustive search.

Try all possible parenthesisations and compare them. Correct, but extremely slow; running time is $\Omega(3^n)$.

Approach 2: Greedy algorithm.

Always do the cheapest multiplication first. Does **not** work correctly — sometimes, it returns a parenthesisation that is not optimal:

Example: Consider

$$\begin{array}{ccccc} A_1 & \cdot & A_2 & \cdot & A_3 \\ 3 \times 100 & & 100 \times 2 & & 2 \times 2 \end{array}$$

Solution proposed by greedy algorithm: $A_1 \cdot (A_2 \cdot A_3)$ with $100 \cdot 2 \cdot 2 + 3 \cdot 100 \cdot 2 = 1000$ multiplications.

Optimal solution: $(A_1 \cdot A_2) \cdot A_3$ with $3 \cdot 100 \cdot 2 + 3 \cdot 2 \cdot 2 = 612$ multiplications.

Solution Attempts (cont'd)

Approach 3: Alternative greedy algorithm.

Set outermost parentheses such that cheapest multiplication is done last.

Doesn't work correctly either (Exercise!).

Approach 4: Recursive (Divide and Conquer).

Divide:

$$(A_1 \cdots A_k) \cdot (A_{k+1} \cdots A_n)$$

For all k, recursively solve the two sub-problems and then take best overall solution.

For $1 \leq i \leq j \leq n$, let

$m[i, j] =$ least number of multiplications needed to compute $A_i \cdots A_j$

Then

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{1 \leq k < j} \left(m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \right) & \text{if } i < j. \end{cases}$$

Analysis of the Recursive Algorithm

Running time $T(n)$ satisfies the recurrence

$$T(n) = \sum_{k=1}^{n-1} \big(T(k) + T(n-k)\big) + \Theta(n).$$

This implies

$$T(n) = \Omega(2^n).$$

Dynamic Programming Solution

As before:

$m[i,j] =$ least number of multiplications needed to compute $A_i \cdots A_j$

Moreover,

$s[i,j] =$ (the smallest) k such that $i \leq k < j$ and
$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j.$$

$s[i,j]$ can be used to reconstruct the optimal parenthesisation.

Idea

Compute the $m[i,j]$ and $s[i,j]$ in a bottom-up fashion.

Implementation

Algorithm MATRIX-CHAIN-ORDER(p)

```
n ← p.length − 1
for i ← 1 to n do
        m[i, i] ← 0
for ℓ ← 2 to n do
        for i ← 1 to n − ℓ + 1 do
                j ← i + ℓ − 1
                m[i, j] ← ∞
                for k ← i to j − 1 do
                        q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                        if q < m[i, j] then
                                m[i, j] ← q
                                s[i, j] ← k
return s
```

Running Time: $\Theta(n^3)$

Example

$$\begin{array}{ccccccc} A_1 & \cdot & A_2 & \cdot & A_3 & \cdot & A_4 \\ 30 \times 1 & & 1 \times 40 & & 40 \times 10 & & 10 \times 25 \end{array}$$

Solution for m and s

m	1	2	3	4
1	0	1200	700	1400
2		0	400	650
3			0	10 000
4				0

s	1	2	3	4
1		1	1	1
2			2	3
3				3
4				

Optimal Parenthesisation

$$A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$$

Multiplying the Matrices

Algorithm MATRIX-CHAIN-MULTIPLY(A, p)

1. $n \leftarrow A$.*length*
2. $s \leftarrow$ MATRIX-CHAIN-ORDER(p)
3. **return** REC-MULT$(A, s, 1, n)$

Algorithm REC-MULT(A, s, i, j)

1. **if** $i < j$ **then**
2. $\quad C \leftarrow$ REC-MULT$(A, s, i, s[i, j])$
3. $\quad D \leftarrow$ REC-MULT$(A, s, s[i, j] + 1, j)$
4. $\quad$ **return** $(C) \cdot (D)$
5. **else**
6. $\quad$ **return** A_i

Reading Assignment

see Wikipedia:

`http://en.wikipedia.org/wiki/Dynamic_programming`

[CLRS] Sections 15.2-15.4 (pages 331-356) *This is Sections 16.1-16.3 (pages 302-320) of [CLR].*

Problems

1. Review the Edit-Distance Algorithm (Inf2B cwk 3 in 06/07) and try to understand why it is a dynamic programming algorithm.
2. Exercise 15.2-1, p.338 of [CLRS] or 16.1-1, p.308 of [CLR].