Algorithms and Data Structures: Dynamic Programming; Matrix-chain multiplication

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

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

A recursive algorithm

Algorithm $\textsc{Rec-Fib}(n)$

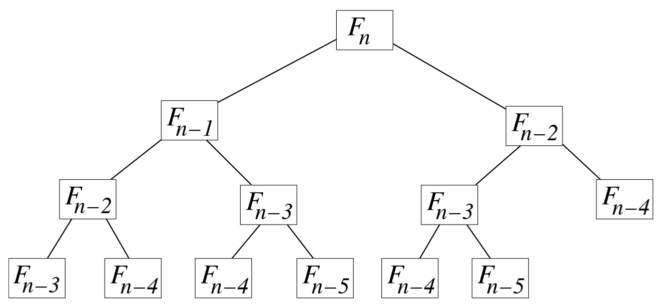
if $n = 0$ **then**
    return 0
else if $n = 1$ **then**
    return 1
else
    return $\textsc{Rec-Fib}(n-1) + \textsc{Rec-Fib}(n-2)$

Ridiculously slow: **exponentially many** repeated computations of $\textsc{Rec-Fib}(j)$ for small values of j.

Fibonacci Example (cont'd)

Why is the recursive solution so slow?
Running time $T(n)$ satisfies

$$T(n) = T(n-1) + T(n-2) + \Theta(1) \geq F_n \sim (1.618)^n.$$



Lower bounds (in order of increasing quality and effort to prove).

1. Let $T'(n) = 2 * T'(n-2) + \Theta(1)$. Show by induction on n that T(n) $\geq T'(n)$. Recursion reaches zero and ends after $n/2$ steps. Thus $T'(n) \geq 2^{n/2} = \sqrt{2}^n \sim (1.41)^n$.

2. We show $F_n \geq \frac{1}{2}(3/2)^n$ for $n \geq 8$ by induction on n. Induction step: $T(n) \geq T(n-1) + T(n-2) \geq \frac{1}{2}((3/2)^{n-1} + (3/2)^{n-2}) = \frac{1}{2}(3/2)^{n-2}((3/2)+1) > \frac{1}{2}(3/2)^{n-2}(3/2)^2 = \frac{1}{2}(3/2)^n$.

3. Let $T'(n) = T'(n-1) + T'(n-2)$ for $n \geq 2$ and $T'(0) = 0$ and $T'(1) = 1$. Then $T(n) \geq T'(n)$. We have

$$\begin{bmatrix} T'(n) \\ T'(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} T'(n-1) \\ T'(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} T'(1) \\ T'(0) \end{bmatrix}$$

Basic linear algebra. Compute eigenvectors and a base transform to diagonalize the matrix. Yields $T'(n) = \Omega((\frac{1+\sqrt{5}}{2})^n)$.

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** $F[n]$

Build "from the bottom up"

Running Time

$$\Theta(n)$$

Very fast in practice - just need an array (of linear size) to store the $\mathrm{F}(i)$ values.
Further improvement to use $\Theta(1)$ space (but still $\Theta(n)$ time): Just use variables to store the current and two previous F_i.

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

A	$\cdot$	B	$\cdot$	C	$\cdot$	D
30×1		1×40		40×10		10×25

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 an optimal parenthesisation to realise this

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

Solution "Attempts"

Approach 1: Exhaustive search (CORRECT but SLOW).
Try all possible parenthesisations and compare them. Correct, but extremely slow. Similar recurrence as Divide and Conquer (see below), thus exponential. See also Textbook.

Approach 2: Greedy algorithm (INCORRECT).
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 (INCORRECT).
Set outermost parentheses such that cheapest multiplication is done last.

Doesn't work correctly either (Exercise!).

Approach 4: Recursive (Divide and Conquer) - (SLOW - **see over**).
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 $1 \le i \le j \le n$, let

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

Then

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

The Recursive Algorithm (SLOW)

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

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

We show $T(n) \ge c2^n$ for some constant c by induction on n. Base case easy (choose constant suitably).
Induction hypothesis $T(n) \ge c2^n$ for some constant c.
Ind. step.: $T(n) \ge \sum_{k=1}^{n-1} \big(T(k) + T(n-k)\big) = \sum_{k=1}^{n-1} \big(2T(k)\big) \ge \sum_{k=1}^{n-1} \big(2c2^k\big) = c\sum_{k=1}^{n-1} \big(2^{k+1}\big) \ge c2^n$.

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 $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$.

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

Idea

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

TURN RECURSION UPSIDE DOWN :-)

Implementation

Algorithm MATRIX-CHAIN-ORDER(p)

$n \leftarrow p.length - 1$
for $i \leftarrow 1$ **to** n **do**
$m[i,i] \leftarrow 0$
for $\ell \leftarrow 2$ **to** n **do**
for $i \leftarrow 1$ **to** $n - \ell + 1$ **do**
$j \leftarrow i + \ell - 1$
$m[i,j] \leftarrow \infty$
for $k \leftarrow i$ **to** $j-1$ **do**
$q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
if $q < m[i,j]$ **then**
$m[i,j] \leftarrow q$
$s[i,j] \leftarrow k$
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)

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

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

if $i < j$ **then**
$C \leftarrow$ REC-MULT($A, s, i, s[i,j]$)
$D \leftarrow$ REC-MULT($A, s, s[i,j]+1, j$)
return $(C) \cdot (D)$
else
return A_i

Problems

See Wikipedia:
http://en.wikipedia.org/wiki/Dynamic_programming
[CLRS] Sections 15.2-15.3

1. Review the Edit-Distance Algorithm and try to understand why it is a dynamic programming algorithm.
2. Exercise 15.2-1 of [CLRS].