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

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2} \quad (\text{for } n \geq 2).$$

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

$$T(n) = T(n-1) + T(n-2) + \Theta(1) \geq F_n \sim (1.618)^n.$$
Lower bounds (in order of increasing quality and effort to prove).

1. Let $T'(n) = 2 \cdot T'(n-2) + \Theta(1)$. 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 $T'(n) = \Omega((\frac{1+\sqrt{5}}{2})^n)$.

ADS: lects 12 and 13 – slide 5 –

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?

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 $n$
4. \hspace{1em} $F[i] \leftarrow F[i-1] + F[i-2]$
5. \hspace{1em} 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 $F(i)$ values.
Further improvement to use $\Theta(1)$ space (but still $\Theta(n)$ time): Just use variables to store the current and two previous $F_i$.

ADS: lects 12 and 13 – slide 6 –

Example

Compute

\[
\begin{array}{cccc}
A & B & C & D \\
30 \times 1 & 1 \times 40 & 40 \times 10 & 10 \times 25
\end{array}
\]

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.

ADS: lects 12 and 13 – slide 7 –

ADS: lects 12 and 13 – slide 8 –
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” (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 $1 \leq i \leq j \leq n$, let

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

Then

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

The Recursive Algorithm (SLOW)

Running time $T(n)$ satisfies the recurrence

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

This implies

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

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

As before:

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

Moreover,

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

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

Idea

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

\( \text{TURN RECURSION UPSIDE DOWN :-)} \)

Implementation

Algorithm \text{Matrix-Chain-Order}(p)

1. \( n \leftarrow p.\text{length} - 1 \)
2. \text{for } i \leftarrow 1 \text{ to } n \text{ do}
3. \( m[i, i] \leftarrow 0 \)
4. \text{for } \ell \leftarrow 2 \text{ to } n \text{ do}
5. \text{for } i \leftarrow 1 \text{ to } n - \ell + 1 \text{ do}
6. \( j \leftarrow i + \ell - 1 \)
7. \( m[i, j] \leftarrow \infty \)
8. \text{for } k \leftarrow i \text{ to } j - 1 \text{ do}
9. \( q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j \)
10. \text{if } q < m[i, j] \text{ then}
11. \( m[i, j] \leftarrow q \)
12. \( s[i, j] \leftarrow k \)
13. \text{return } s

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

Example

\[
\begin{array}{cccc}
A_1 & A_2 & A_3 & A_4 \\
30 \times 1 & 1 \times 40 & 40 \times 10 & 10 \times 25
\end{array}
\]

Solution for \( m \) and \( s \)

\[
\begin{array}{cccc|cccc}
& 1 & 2 & 3 & 4 \\
m & 1 & 0 & 1200 & 700 & 1400 \\
& 2 & 0 & 400 & 650 \\
& 3 & 0 & 10000 \\
& 4 & 0 & 0
\end{array}
\begin{array}{cccc}
s & 1 & 2 & 3 & 4 \\
1 & 1 & 1 & 1 \\
2 & 2 & 3 \\
3 & 3 \\
4 & 4
\end{array}
\]

Optimal Parenthesisation

\[ A_1 \cdot ((A_2 \cdot A_3) \cdot A_4) \]

Multiplying the Matrices

Algorithm \text{Matrix-Chain-Multiply}(A, p)

1. \( n \leftarrow A.\text{length} \)
2. \( s \leftarrow \text{Matrix-Chain-Order}(p) \)
3. \text{return } \text{Rec-Mult}(A, s, 1, n)

Algorithm \text{Rec-Mult}(A, s, i, j)

1. \text{if } i < j \text{ then}
2. \( C \leftarrow \text{Rec-Mult}(A, s, i, s[i, j]) \)
3. \( D \leftarrow \text{Rec-Mult}(A, s, s[i, j] + 1, j) \)
4. \text{return } (C) \cdot (D)
5. \text{else}
6. \text{return } A_i
Problems

See Wikipedia:
[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].