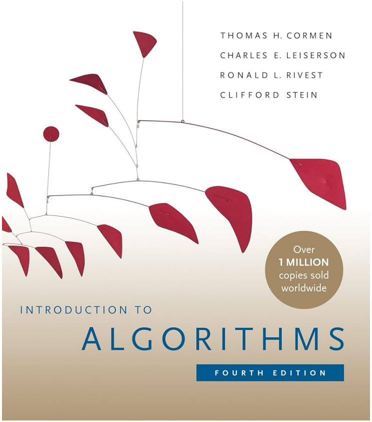
Introduction to Algorithms
Lecture 7: DP Exercises

Cormen, Leiserson, Rivest, and Stein. MIT Press. 2022.

April 22, 2025

Introduction to Algorithms



Content has been extracted from *Introduction to Algorithms*, Fourth Edition, by Cormen, Leiserson, Rivest, and Stein. MIT Press. 2022.

Visit `https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/`.

Plan

Matrix-chain multiplication

Optimal binary search trees

Matrix-chain multiplication

Problem: Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, compute the product $A_1 A_2 \cdots A_n$ using standard matrix multiplication (not Strassen's method) while minimizing the number of scalar multiplications.

How to parenthesize the product to minimize the number of scalar multiplications?

Matrix-chain multiplication

- Suppose multiplying matrices A and B: $C = A \cdot B$.
- The matrices must be compatible: number of columns of A equals number of rows of B.
- If A is $p \times q$ and B is $q \times r$, then C is $p \times r$ and takes pqr scalar multiplications.

Example

$A_1 : 10 \times 100$, $A2 : 100 \times 5$, $A3 : 5 \times 50$. Compute $A_1 A_2 A_3$, which is 10×50.

- Try parenthesizing by $((A_1 A_2) A_3)$. First perform $10 \times 100 \times 5 = 5000$ multiplications, then perform $10 \times 5 \times 50 = 2500$, for a total of 7500.
- Try parenthesizing by $(A_1 (A_2 A_3))$. First perform $100 \times 5 \times 50 = 25,000$ multiplications, then perform $10 \times 100 \times 50 = 50,000$, for a total of 75,000.
- The first way is 10 times faster.

Input to the Problem

- Let A_i be $p_{i-1} \times p_i$. The input is the sequence of dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$.
- **Note**: Not actually multiplying matrices. Just deciding an order with the lowest cost.

Counting the Number of Parenthesizations

- Let $P(n)$ denote the number of ways to parenthesize a product of n matrices. $P(1) = 1$.
- When $n \geq 2$, can split anywhere between A_k and A_{k+1} for $k = 1, 2, \ldots, n-1$. Then have to split the subproducts.
- Get

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

- The solution is $P(n) = \Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$. So brute force is a bad strategy.

Step 1: Structure of an optimal solution

- Let $A_{i:j}$ be the matrix product $A_i A_{i+1} \cdots A_j$.
- If $i < j$, then must split between A_k and A_{k+1} for some $i \le k < j \Rightarrow$ compute $A_{i:k}$ and $A_{k+1:j}$ and then multiply them together.
- Cost is

$$\begin{aligned} & \text{cost of computing } A_{i:k} \\ + & \text{ cost of computing } A_{k+1:j} \\ + & \text{ cost of multiplying them together.} \end{aligned}$$

Optimal substructure

- Suppose that optimal parenthesization of $A_{i:j}$ splits between A_k and A_{k+1}.
- Then the parenthesization of $A_{i:k}$ must be optimal.
- Otherwise, if there's a less costly way to parenthesize it, you'd use it and get a parenthesization of $A_{i:j}$ with a lower cost. Same for $A_{k+1:j}$.
- Therefore, to build an optimal solution to $A_{i:j}$:
  - split it into how to optimally parenthesize $A_{i:k}$ and $A_{k+1:j}$,
  - find optimal solutions to these subproblems,
  - and then combine the optimal solutions.
- Need to consider all possible splits.

Step 2: A recursive solution

- Define the cost of an optimal solution recursively in terms of optimal subproblem solutions.
- Let $m[i, j]$ be the minimum number of scalar multiplications to compute $A_{i:j}$. For the full problem, want $m[1, n]$.
  - If $i = j$, then just one matrix $m[i, i] = 0$ for $i = 1, 2, \ldots, n$.
  - If $i < j$, then suppose the optimal split is between A_k and A_{k+1}, where $i \leq k < j$.
  - Then $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_ip_j$.

Step 2: A recursive solution

- But that's assuming you know the value of k. Have to try all possible values and pick the best, so that

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

- That formula gives the cost of an optimal solution, but not how to construct it.
  - Define $s[i,j]$ to be a value of k to split $A_{i:j}$ in an optimal parenthesization.
  - Then $s[i;j] = k$ such that $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.

Step 3: Compute the optimal costs

- Could implement a recursive algorithm based on the above equation for $m[i, j]$ but it would take exponential time.
- There are not all many subproblems:
  - just one for each i, j such that $1 \leq i \leq j \leq n$.
  - there are $\binom{n}{2} + n = \Theta(n^2)$ of them.
  - thus, a recursive algorithm would solve the same subproblem over and over.
- In other words, this problem has overlapping subproblems.

Step 3: Compute the optimal costs

- Here is a tabular, bottom-up method to solve the problem.
- It solves subproblems in order of increasing chain length.

```
MATRIX-CHAIN-ORDER(p, n)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                          // chain length 1
    m[i, i] = 0
for l = 2 to n                          // l is the chain length
    for i = 1 to n − l + 1              // chain begins at A_i
        j = i + l − 1                   // chain ends at A_j
        m[i, j] = ∞
        for k = i to j − 1              // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q             // remember this cost
                s[i, j] = k             // remember this index
return m and s
```

Step 3: Compute the optimal costs

- Here is a tabular, bottom-up method to solve the problem.
- It solves subproblems in order of increasing chain length.

```
MATRIX-CHAIN-ORDER(p, n)
let m[1:n, 1:n] and s[1:n-1, 2:n] be new tables
for i = 1 to n                          // chain length 1
    m[i,i] = 0
for l = 2 to n                          // l is the chain length
    for i = 1 to n - l + 1              // chain begins at A_i
        j = i + l - 1                   // chain ends at A_j
        m[i,j] = ∞
        for k = i to j - 1              // try A_{i:k} A_{k+1:j}
            q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
            if q < m[i,j]
                m[i,j] = q              // remember this cost
                s[i,j] = k              // remember this index
return m and s
```

Time: $O(n^3)$, from triply nested loops. Also $\Omega(n^3) \Rightarrow \Theta(n^3)$.

Step 4: Construct an optimal solution

- With the s table filled in, recursively print an optimal solution.
- Initial call is PRINT-OPTIMAL-PARENS$(s, 1, n)$.

```
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 ")"
```

Plan

Matrix-chain multiplication

Optimal binary search trees

Optimal binary search trees

- Given sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys, sorted ($k_1 < k_2 < \ldots < k_n$).
- Want to build a binary search tree from the keys.
- For k_i, have probability p_i that a search is for k_i.
- Want BST with minimum expected search cost.

BST Cost

Actual cost = # of items examined.

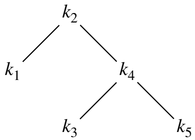
For k_i, $cost = depth_T(k_i) + 1$, where $depth_T(k_i)$ = depth k_i in BST T.

$E[$ search cost in $T]$

$$
\begin{aligned}
&= \sum_{i=n}^{n} (depth_T(k_i) + 1) \cdot p_i \\
&= \sum_{i=n}^{n} depth_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i \\
&= 1 + \sum_{i=n}^{n} depth_T(k_i) \cdot p_i
\end{aligned}
$$

Example

i	1	2	3	4	5
p_i	.25	.2	.05	.2	.3

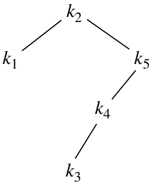


i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	2	.1
4	1	.2
5	2	.6
		1.15

- Therefore, E[search cost] $= 2.15$.

Example

i	1	2	3	4	5
p_i	.25	.2	.05	.2	.3



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	3	.15
4	2	.4
5	1	.3
		1.10

- Therefore, E[search cost] $= 2.10$, which turns out to be optimal.

Observations

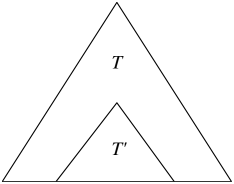
- Optimal BST might not have smallest height.
- Optimal BST might not have highest-probability key at root.

Build by exhaustive checking?

- Construct each n-node BST.
- For each, put in keys.
- Then compute expected search cost.
- But there are $\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$ different BSTs with n nodes.

Step 1: The structure of an optimal BST

Consider any subtree of a BST. It contains keys in a contiguous range $k_i, \ldots, k_j$ for some $1 \le i \le j \le n$.

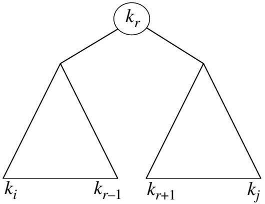


If T is an optimal BST and T contains subtree T' with keys $k_i, \ldots, k_j$, then T' must be an optimal BST for keys $k_i, \ldots, k_j$.

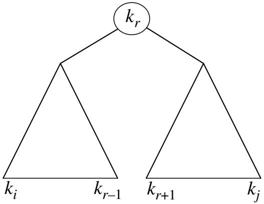
Step 1: The structure of an optimal BST

Use optimal substructure to construct an optimal solution to the problem from optimal solutions to subproblems:

- Given keys $k_i, \ldots, k_j$ (the problem).
- One of them, k_r, where $i \leq r \leq j$, must be the root.
- Left subtree of k_r contains $k_i, \ldots, k_{r-1}$.
- Right subtree of k_r contains $k_{r+1}, \ldots, k_j$.



Step 1: The structure of an optimal BST



- If
  - you examine all candidate roots k_r, for $i \leq r \leq j$, and
  - you determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$,

  then you're guaranteed to find an optimal BST for $k_i, \ldots, k_j$.

Step 2: Recursive solution

Subproblem domain:

- Find optimal BST for $k_i, \ldots, k_j$, where $i \geq 1$, $j \leq n$, $j \geq i - 1$.
- When $j = i - 1$, the tree is empty.

Define $e[i, j]$ = expected search cost of optimal BST for $k_i, \ldots, k_j$.

Step 2: Recursive solution

- If $j = i - 1$, then $e[i, j] = 0$.
- If $j \geq i$,
  - Select root k_r, for some $i \leq r \leq j$.
  - Make an optimal BST with $k_i, \ldots, k_{r-1}$ as the left subtree.
  - Make an optimal BST with $k_{r+1}, \ldots, k_j$ as the right subtree.
  - Note: when $r = i$, left subtree is $k_i, \ldots, k_{i-1}$; when $r = j$, right subtree is $k_{j+1}, \ldots, k_j$. These subtrees are empty.

Step 2: Recursive solution

When a subtree becomes a subtree of a node:

- Depth of every node in subtree goes up by 1.
- Expected search cost increases by

$$w(i, j) = \sum_{l=i}^{j} p_l$$

If k_r is the root of an optimal BST for $k_i, \ldots, k_j$:

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$$

- But $w(i, j) = w(i, r-1) + p_r + w(r+1, j)$.
- Therefore, $e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j)$

Step 2: Recursive solution

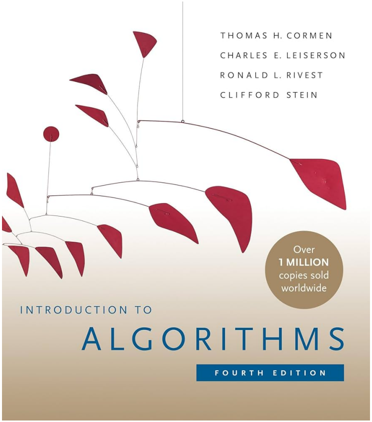
That equation assumes that we already know which key is k_r. We don't.

- Try all candidates, and pick the best one:

$$e[i,j] = \begin{cases} 0 & \text{if } j = i-1, \\ \min\{e[i,r-1] + e[r+1,j] + w(i,j) : i \le r \le j\} & \text{if } i \le j. \end{cases}$$

- Could write a recursive algorithm. . .

Introduction to Algorithms



Content has been extracted from *Introduction to Algorithms*, Fourth Edition, by Cormen, Leiserson, Rivest, and Stein. MIT Press. 2022.

Visit `https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/`.