

CS624 - Analysis of Algorithms

Dynamic Programming

October 8, 2019

Problem – Making Change

- Task – buy a cup of coffee (say it costs 63 cents).
- You are given an unlimited number of coins of all types (neglect 50 cents and 1 dollar).
- Pay exact change.
- What is the combination of coins you'd use?



1 cent



5 cents



10 cents



25 cents



Greedy Thinking – Change Making

- Logically, we want to minimize the number of coins.
- The problem is then: Count change using the fewest number of coins – we have 1, 5, 10, 25 unit coins to work with.
- The "greedy" part lies in the order: We want to use as many large-value coins to minimize the total number.
- When counting 63 cents, use as many 25s as fit, $63 = 2(25) + 13$, then as many 10s as fit in the remainder: $63 = 2(25) + 1(10) + 3$, no 5's fit, so we have $63 = 2(25) + 1(10) + 3(1)$, 6 coins.

Greedy Algorithms

- A greedy person grabs everything they can as soon as possible.
- Similarly a greedy algorithm makes locally optimized decisions that appear to be the best thing to do at each step.
- Example: Change-making greedy algorithm for “change” amount, given many coins of each size:
 - Loop until $\text{change} == 0$:
 - Find largest-valued coin less than change, use it.
 - $\text{change} = \text{change} - \text{coin-value}$;

Change Making

- The greedy method gives the optimal solution for US coinage.
- With different coinage, the greedy algorithm doesn't always find the optimal solution.
- Example of a coinage with an additional 21 cent piece. Then $63 = 3(21)$, but greedy says use 2 25s, 1 10, and 3 1's, a total of 6 coins.
- The coin values need to be spread out enough to make greedy work.
- But even some spread-out cases don't work. Consider having pennies, dimes and quarters, but no nickels.
- Then 30 by greedy uses 1 quarter and 5 pennies, ignoring the best solution of 3 dimes.

Greedy Algorithms

- Greedy algorithms are very popular
- They do not always guarantee the optimal solution but they are often simple and can be used as approximation algorithms when the exact solution is too hard.
- Sometimes they give the optimal solution, as with the US coins above.
- We will visit greedy algorithms later in the course.
- For now we need a method that guarantees optimality for any coin combination.

(Very bad) Recursive Solution

Example: change for 63 cents with coins = {25, 10, 5, 1, 21} no order required in array.

```
makeChange(63)
minCoins = 63
loop over j from 1 to 63/2 = 31
    thisCoins = makeChange(j) + makeChange(63-j)
    if thisCoins < minCoins
        minCoins = thisCoins
return minCoins
```

Lots and lots of redundant calls!

(Very bad) Recursive Solution

$$T(n) = T(n-1) + T(n-2) + T(n-3) + \dots + T(n/2) + \dots$$

Incredibly bad, right?

- We know we have 1,5,10,21 and 25.
- Therefore, the optimal solution must be the minimum of the following:
 - 1 (A 1 cent) + optimal solution for 62.
 - 1 (A 5 cent) + optimal solution for 58.
 - 1 (A 10 cent) + optimal solution for 53.
 - 1 (A 21 cent) + optimal solution for 42.
 - 1 (A 25 cent) + optimal solution for 38.
 - This reduces the number of recursive calls drastically.
- Naive implementation still makes lots of redundant calls.

Dynamic Programming Implementation

- Idea – hold on to the fact that you only have to look at five previous solutions
- But instead of performing the same calculation over and over again, save pre-calculated results to an array.
- The answer to a large change depends only on results of smaller calculations, so we can calculate the optimal answer for all the smaller change and save it to an array.
- Then go over the array and minimize on:
 - $change(N) = \min_{k \in K} \{change(N - k) + 1\}$
 - For all K types of coins, in our example $K = \{1, 5, 10, 21, 25\}$
- Runtime – $O(N * K)$.

Dynamic Programming for Change Problem

```
public static void makeChange( int [ ] coins, int differentCoins,
                               int maxChange, int [ ] coinsUsed, int [ ] lastCoin )
{
    coinsUsed[ 0 ] = 0; lastCoin[ 0 ] = 1;
    for( int cents = 1; cents <= maxChange; cents++ ) {
        int minCoins = cents;
        int newCoin = 1;
        for( int j = 0; j < differentCoins; j++ ) {
            if( coins[ j ] > cents ) // Cannot use coin j
                continue;
            if( coinsUsed[ cents - coins[ j ] ] + 1 < minCoins ) {
                minCoins = coinsUsed[ cents - coins[ j ] ] + 1;
                newCoin = coins[ j ];
            }
        }
        coinsUsed[ cents ] = minCoins;
        lastCoin[ cents ] = newCoin;
    }
}
```

Dynamic Programming

- An algorithm design technique for **optimization problems**: often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent and may share subsubproblems,
- However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)
- 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.
- **Key:** determine structure of optimal solutions

Longest Common Subsequence (LCS)

Definition

A *subsequence* of a sequence $A = \{a_1, a_2, \dots, a_n\}$ is a sequence $B = \{b_1, b_2, \dots, b_m\}$ (with $m \leq n$) such that

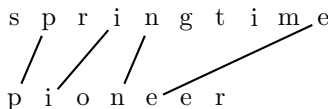
- Each b_i is an element of A .
 - If b_i occurs before b_j in B (i.e., if $i < j$) then it also occurs before b_j in A .
-
- We do *not* assume that the elements of B are consecutive elements of A .
 - For example: “axdy” is a subsequence of “baxefdoym”

The “longest common subsequence” problem is simply this:

Given two sequences $X = \{x_1, x_2, \dots, x_m\}$ and $Y = \{y_1, y_2, \dots, y_n\}$ (note that the sequences may have different lengths), find a subsequence common to both whose length is longest.

LCS – example

s p r i n g t i m e
p i o n e e r



- This is part of a class of what are called *alignment problems*, which are extremely important in biology.
- It can help us to compare genome sequences to deduce quite accurately how closely related different organisms are, and to infer the real “tree of life”.
- Trees showing the evolutionary development of classes of organisms are called “phylogenetic trees”.
- A lot of this kind of comparison amounts to finding common subsequences.

LCS – Naive approach

- Try the obvious approach: list all the subsequences of X and check each to see if it is a subsequence of Y , and pick the longest one that is.
- There are 2^m subsequences of X . To check to see if a subsequence of X is also a subsequence of Y will take time $O(n)$. (Is this obvious?)
- Picking the longest one an $O(1)$ job, since we can keep track as we proceed of the longest subsequence that we have found so far.
- So the cost of this method is $O(n2^m)$.
- That's pretty awful, since the strings that we are concerned with in biology have hundreds or thousands of elements *at least*.

LCS – Optimal Substructure

- We have two strings, with possibly different lengths:
 $X = \{x_1, x_2, \dots, x_m\}$ and $Y = \{y_1, y_2, \dots, y_n\}$
- A *prefix* of a string is an initial segment. So we define for each i less than or equal to the length of the string the prefix of length i :
 $X = \{x_1, x_2, \dots, x_i\}$ and $Y = \{y_1, y_2, \dots, y_i\}$
- A solution of our problem reflects itself in solutions of prefixes of X and Y .

Theorem

Let $Z = \{z_1, z_2, \dots, z_k\}$ be any LCS of X and Y .

- 1 If $x_m = y_n$, then $z_k = x_m = y_n$, and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .
- 2 If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y .
- 3 If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1} .

Proof.

- ① By assumption $x_m = y_n$. If z_k does not equal this value, then Z must be a common subsequence of X_{m-1} and Y_{n-1} , and so the sequence $Z' = \{z_1, z_2, \dots, z_k, x_m\}$ would be a common subsequence of X and Y . But this is a longer common subsequence than Z , and this is a contradiction.
- ② If $z_k \neq x_m$, then Z must be a subsequence of X_{m-1} , and so it is a common subsequence of X_{m-1} and Y . If there were a longer one, then it would also be a common subsequence of X and Y , which would be a contradiction.
- ③ This is really the same as 2.



Corollary

If $x_m \neq y_n$, then either

- Z is an LCS of X_{m-1} and Y , or
- Z is an LCS of X and Y_{n-1} .

- Thus, the LCS problem has what is called the *optimal substructure property*: a solution contains within it the solutions to subproblems – in this case, to subproblems constructed from prefixes of the original data.
- This is one of the two keys to the success of a dynamic programming solution.

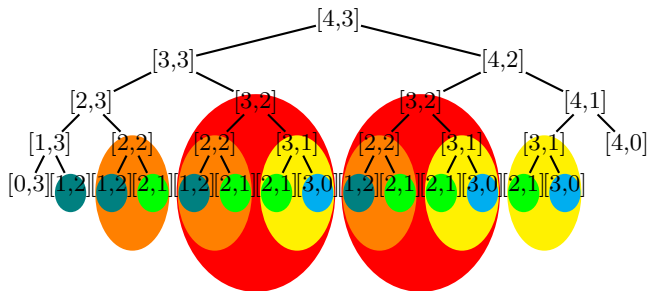
Recursive Algorithm

- Let $c[i, j]$ be the length of the LCS of X_i and Y_j . Based on The optimal substructure theorem, we can write the following recurrence:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i - 1, j], c[i, j - 1]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- The optimal substructure property allows us to write down an elegant recursive algorithm.
- However, the cost is still far too great – we can see that there are $\Omega(2^{\min\{m, n\}})$ nodes in the tree, which is still a killer.

Recursive Algorithm



Overlapping Substructures

- There are only $O(mn)$ distinct nodes, but many nodes appear multiple times.
- We only have to compute each subproblem once, and save the result so we can use it again.
- This is called *memoization*, which refers to the process of saving (i.e., making a “memo”) of a intermediate result so that it can be used again without recomputing it.
- Of course the words “memoize” and “memorize” are related etymologically, but they are different words, and you should not mix them up.

Algorithm 1 LCSLength(X, Y, m, n)

```
1: for  $i \leftarrow 1 \dots m$  do
2:    $c[i, 0] \leftarrow 0$ 
3: end for
4: for  $j \leftarrow 0 \dots n$  do
5:    $c[0, j] \leftarrow 0$ 
6: end for
7: for  $i \leftarrow 1 \dots m$  do
8:   for  $j \leftarrow 1 \dots n$  do
9:     if  $x_i == y_j$  then
10:       $c[i, j] \leftarrow c[i - 1, j - 1] + 1$ ;  $b[i, j] \leftarrow \nwarrow$ 
11:     else
12:       if  $c[i - 1, j] \geq c[i, j - 1]$  then
13:          $c[i, j] \leftarrow c[i - 1, j]$ ;  $b[i, j] \leftarrow \uparrow$ 
14:       else
15:          $c[i, j] \leftarrow c[i, j - 1]$ ;  $b[i, j] \leftarrow \leftarrow$ 
16:       end if
17:     end if
18:   end for
19: end for
20: return  $c$  and  $b$ 
```

LCS Table – Example

j	0	1	2	3	4	5	6
$i \backslash y_j$		B	D	C	A	B	A
0 x_i		0	0	0	0	0	0
1 A		0	↑	↑	↖	1	↖
2 B		↖	1	←	1	↑	↖
3 C		↑	↑	↖	2	←	2
4 B		↖	1	↑	↑	↑	↖
5 D		↑	↖	2	↑	↑	↑
6 A		↑	↑	↑	↖	3	↖
7 B		↖	↑	↑	↑	↖	↑

Constructing the Actual LCS

Just backtrack from $c[m, n]$ following the arrows:

Algorithm 2 PrintLCS(b, X, i, j)

```
1: if  $i = 0$  or  $j = 0$  then
2:   return
3: end if
4: if  $b[i, j] == \text{"↖"}$  then
5:   PrintLCS( $b, X, i - 1, j - 1$ )
6:   PRINT  $x_i$ 
7: else
8:   if  $b[i, j] == \text{"↑"}$  then
9:     PrintLCS( $b, X, i - 1, j$ )
10:  else
11:    PrintLCS( $b, X, i, j - 1$ )
12:  end if
13: end if
```

What Makes Dynamic Programming Work?

It is important to understand the two properties of this problem that made it possible for use of dynamic programming:

- Optimal substructure: subproblems are just “smaller versions” of the main problem.
- Finding the LCS of two substrings could be reduced to the problem of finding the LCS of shorter substrings.
- This property enables us to write a recursive algorithm to solve the problem, but this recursion is much too expensive – typically, it has an exponential cost.
- Overlapping subproblems: This is what saves us: The same subproblem is encountered many times, so we can just solve each subproblem once and “memoize” the result.
- In the current problem, that memoization cut down the cost from exponential to quadratic, a dramatic improvement.

Optimal Binary Search Tree

- Given sequence $K = k_1 < k_2 < \dots < k_n$ of n sorted keys, with a search probability p_i for each key k_i .
- Want to build a binary search tree (BST) with minimum expected search cost.
- Actual cost = # of items examined.
- For key k_i , $cost = depth_T(k_i) + 1$, where $depth_T(k_i)$ = depth of k_i in BST T .
- Example – dictionary search, where not all words have equal probability to be searched.

Example

- Suppose we have a BST containing 5 words.
- We can create an additional 6 “dummy” nodes to represent searches for words not in the tree, like this (where we have arranged the words in alphabetical order:

d_0 k_1 d_1 k_2 d_2 k_3 d_3 k_4 d_4 k_5 d_5

The following table shows the probabilities of searching for these different nodes:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Cost of Searching a BST

- p_i is the probability of searching for k_i (the probability of searching for the i^{th} word)
- q_i (for $i \geq 1$) is the probability of searching for d_i (the probability of searching for a word between the i^{th} word and the $(i+1)^{\text{th}}$ word in the tree)
- q_0 is the probability of searching for a word before the first word in the tree

Of course we must have

$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$$

Expected Search Cost of Each Tree

- Assume that the cost of a search is the number of nodes visited in the search.
- Denote the expected search cost for a tree T by $E(T)$.
- For any node x in the tree T , let us say that $\text{depth}_T(x)$ is the distance of x from the root of T . (So the root has depth 0.)
- Then we have

$$E(T) = \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i) + 1) \cdot q_i$$

- Note that this can also be written as

$$E(T) = \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{depth}_T(d_i) \cdot q_i + \left(\sum_{i=1}^n p_i + \sum_{i=0}^n q_i \right)$$

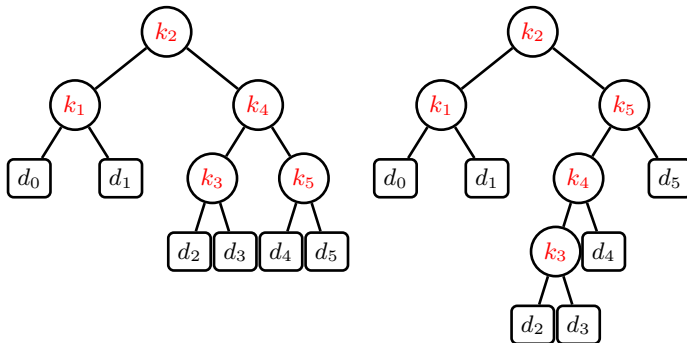
Expected Search Cost of Each Tree

- If we think of the probabilities p_i and q_i as “weights”, then the total weight of the tree is $w(1, n) = \sum_{i=1}^n p_i + \sum_{i=0}^n q_i$
- You will see below why we called this $w(1, n)$, and not just w .
- So the equation above could also be written like this:

$$E(T) = w(1, n) + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{depth}_T(d_i) \cdot q_i$$

- As it happens, we know that $w(1, n)$ actually equals 1, but we will write some similar identities below in which this is no longer true.

Example – Two Trees with 5 Keys



Example – Expected Search Cost of Left Tree

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

The expected search cost for the other tree is 2.75. So putting the nodes of maximum probability highest is not necessarily the best thing to do.

Optimal Substructure

- The number of binary trees on n nodes is

$$\frac{1}{n+1} \binom{2n}{n} = \frac{4^n}{\sqrt{\pi} n^{3/2}} (1 + O(1/n))$$

- Certainly exhaustive search is not a useful way of finding the best tree in this problem.
- Substructure naturally involves subtrees.
- Our problem does exhibit optimal substructure in the following way:
- Since our tree is a BST, any subtree contains a contiguous sequence of keys $\{k_i, \dots, k_j\}$ and its leaves will be the contiguous set of dummy nodes $\{d_{i-1}, \dots, d_j\}$.
- Let us denote the optimal binary search tree containing exactly these nodes by $T_{i,j}$.

Optimal Substructure

The optimal substructure property that our problem possesses is this:

Theorem (Optimal substructure for the optimal binary search tree problem)

If T is an optimal binary search tree and if T' is any subtree of T , then T' is an optimal binary search tree for its nodes.

Proof.

This is a standard cut-and-paste argument. □

Compute the Optimal Solution

- Let $e[i, j]$ be the expected cost of searching an optimal binary search tree containing the keys $\{k_i, \dots, k_j\}$.
- That is, $e[i, j]$ is the expected cost of searching the tree $T_{i,j}$.
- Ultimately, we want to compute $e[1, n]$.
- If the optimal binary search tree for this subproblem has k_r as its root then the problem divides into three parts:
 - The expected cost of searching the tree $T_{i,r-1}$ built from the nodes $\{i, \dots, r-1\}$, adjusted for the fact that this is a subtree of our original tree $T_{i,j}$ and so all the depths should be 1 greater than they are in the subtree.
 - The cost of searching for the root k_r .
 - The expected cost of searching the tree $T_{r+1,j}$ built from the nodes $\{k_{r+1}, \dots, k_j\}$, with all the depths 1 greater than they are in the subtree.

Compute the Optimal Solution

- r can take any of the values $\{i, \dots, j\}$.
- If $r = i$ then the first subtree $T_{i,r-1}$ is empty (rather, we let it contain the dummy node d_{i-1} since there is nowhere else to put that node anyway).
- Similarly, if $r = j$ then the second subtree $T_{r+1,j}$ contains the dummy node d_j .
- In other words – the tree $T_{s,s-1}$ built from the nodes $\{k_s, \dots, k_{s-1}\}$ contains the single node d_{s-1} and its expected cost $e[s, s-1]$ will thus be q_s .
- Set $w(i, j) = \sum_{l=i}^j p_l + \sum_{l=i-1}^j q_l$
- This is the sum of the probabilities of all the nodes in the tree $T_{i,j}$ built from the nodes $\{k_i, \dots, k_j\}$.

Compute the Optimal Solution

- The tree $T_{i,r-1}$ has cost $e[i, r - 1]$, but as a subtree of $T_{i,r}$, its cost has to be increased by increasing each depth number by 1 – this amounts to adding $w(i, r - 1)$.
- The expected cost that the subtree $T_{i,r-1}$ contributes to the expected cost of $T_{i,j}$ is $e[i, r - 1] + w(i, r - 1)$.
- A similar argument applies to the other subtree $T_{r+1,j}$.
- So we get

$$\begin{aligned} e[i, j] &= E(T_{i,j}) \\ &= p_r + (E(T_{i,r-1}) + w(i, r - 1)) \\ &\quad + (E(T_{r+1,j}) + w(r + 1, j)) \\ &= p_r + (e[i, r - 1] + w(i, r - 1)) \\ &\quad + (e[r + 1, j] + w(r + 1, j)) \end{aligned}$$

Simplifying Things a Little

- Note that $w(i, j) = w(i, r - 1) + p_r + w(r + 1, j)$
- So we have $e[i, j] = w(i, j) + e[i, r - 1] + e[r + 1, j]$
- We have to take the minimum over all possible choices of r .
- Thus we have

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 \\ w(i, j) + \min_{i \leq r \leq j} \{e[i, r - 1] + e[r + 1, j]\} & \text{if } i \leq j \end{cases}$$

- We can use it to compute $e[1, n]$.
- However this algorithm is still exponential in cost.
- We can do better because this problem also exhibits the property of *overlapping subproblems*.
- There are only $O(n^2)$ values $e[i, j]$ with $1 \leq i \leq n + 1$ and $0 \leq j \leq n$, so we can memoize the.

More Efficient Calculation

- Store pre-computed values in an array $e[1 \dots n+1, 0 \dots n]$.
- We can also store the values $w(i, j)$ in a table $w[1 \dots n+1, 0 \dots n]$.
- We have

$$w[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 \\ w[i, j - 1] + p_j + q_j & \text{otherwise} \end{cases}$$

- There are $O(n^2)$ values of $w[i, j]$ and each one takes a constant time to compute, so the total cost of computing the w array is $O(n^2)$.
- The cost of computing each value of $e[i, j]$ is $O(n)$ and there are $O(n^2)$ such values, so the cost of computing all the values of $e[i, j]$ is $O(n^3)$.
- So the total cost of computing the w array first and then the e array is $O(n^2) + O(n^3) = O(n^3)$

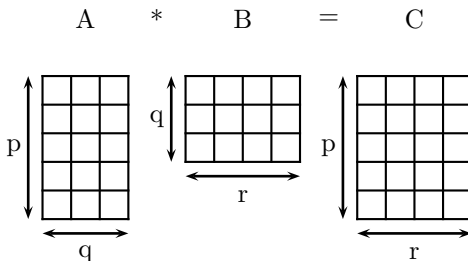
Example – Chain Operations

- Determine the optimal sequence for performing a series of operations (the general class of the problem is important in compiler design for code optimization & in databases for query optimization)
- For example: given a series of matrices: $A_1 \dots A_n$, we can “parenthesize” this expression however we like, since matrix multiplication is associative (**but not commutative**)
- Multiply a pxq matrix by a qxr matrix B, the result will be a pxr matrix C. (# of columns of A must be equal to # of rows of B.)

Matrix Multiplications

$$\text{for } 1 \leq i \leq p \text{ and } 1 \leq j \leq r, C[i,j] = \sum_{k=1}^q A[i,k]B[k,j]$$

Observe that there are pr total entries in C and each takes $O(q)$ time to compute, thus the total time to multiply 2 matrices is pqr .



Chain Matrix Multiplication (CMM)

- Given a sequence of matrices A_1, A_2, \dots, A_n , and dimensions $p_0, p_1 \dots p_n$ where A_i is of dimension $p_{i-1} \times p_i$, determine multiplication sequence that minimizes the number of operations.
- This algorithm does not perform the multiplication, it just figures out the best order in which to perform the multiplication.

CMM – Example

- Consider 3 matrices: A_1 be 5×4 , A_2 be 4×6 , and A_3 be 6×2 .
- Count the number of operations:

$$\text{Mult}[(A_1 A_2) A_3] = (5 \times 4 \times 6) + (5 \times 6 \times 2) = 180$$

$$\text{Mult}[A_1 (A_2 A_3)] = (4 \times 6 \times 2) + (5 \times 4 \times 2) = 88$$

- Even for this small example, considerable savings can be achieved by reordering the evaluation sequence.

CMM – Naive Algorithm

- If we have just 1 item, then there is only one way to parenthesize.
- If we have n items, then there are $n-1$ places where you could break the list with the outermost pair of parentheses, namely just after the first item, just after the 2^{nd} item, etc. and just after the $(n-1)^{th}$ item.
- When we split just after the k^{th} item, we create two sub-lists to be parenthesized, one with k items and the other with $n-k$ items.
- Then we consider all ways of parenthesizing these.
- If there are L ways to parenthesize the left sub-list, R ways to parenthesize the right sub-list, then the total possibilities is $L \cdot R$.

Cost of Naive Algorithm

- The number of different ways of parenthesizing n items is

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

- Specifically $P(n) = C(n-1)$.
- $C(n) = (1/(n+1)) * \binom{2n}{n} = \Omega(4^n/n^{3/2})$

DP Solution (I)

- Let $A_{i...j}$ be the product of matrices i through j . $A_{i...j}$ is a $p_{i-1} \times p_j$ matrix.
- At the highest level, we are multiplying two matrices together. That is, for any k , $1 \leq k \leq n - 1$, $A_{1...n} = (A_{1...k})(A_{k+1...n})$
- The problem of determining the optimal sequence of multiplication is broken up into 2 parts:
 - Q: How do we decide where to split the chain (what k)?
 - A: Consider all possible values of k .
 - Q: How do we parenthesize the subchains $A_{1...k}$ & $A_{k+1...n}$?
 - A: Solve by recursively applying the same scheme.
- NOTE: this problem satisfies the “principle of optimality”
- Next, we store the solutions to the sub-problems in a table and build the table in a bottom-up manner.

- For $1 \leq i \leq j \leq n$, let $m[i,j]$ denote the minimum number of multiplications needed to compute $A_{i...j}$.
- Example: Minimum number of multiplies for $A_{3...7}$

$$A_1 A_2 \underbrace{A_3 A_4 A_5 A_6 A_7}_{m[3,7]} A_8 A_9$$

In terms of p_i , the product $A_{3...7}$ has dimensions $p_2 \times p_7$.

- The optimal cost can be described be as follows:
- $i = j \Rightarrow$ the sequence contains only 1 matrix, so $m[i, j] = 0$.
- $i < j \Rightarrow$ This can be split by considering each k , $i \leq k < j$, as $A_{i...k}(p_{i-1} \times p_k)$ times $A_{k+1...j}(p_k \times p_j)$.
- This suggests the following recursive rule for computing $m[i, j]$:

$$m[i, i] = 0$$

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j) \forall i < j$$

Computing $m[i,j]$

For a specific K :

$$\begin{aligned} & (A_i \dots A_k)(A_{k+1} \dots A_j) \\ = & A_{i \dots k}(A_{k+1} \dots A_j) && (m[i, k] \text{ mults}) \\ = & A_{i \dots k} A_{k+1 \dots j} && (m[k+1, j] \text{ mults}) \\ = & A_{i \dots j} && (p_{i-1} p_k p_j \text{ mults}) \end{aligned}$$

For solution, evaluate for all k and take minimum.

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j)$$

Algorithm 3 MatrixChainOrder(p)

```
1:  $n \leftarrow \text{length}[p] - 1$ 
2: for  $i \leftarrow 1 \dots n$  // initialization:  $O(n)$  time do
3:    $m[i, i] \leftarrow 0$ 
4:   for  $L \leftarrow 2 \dots n$  //  $L = \text{length of sub-chain}$  do
5:     for  $i \leftarrow 1 \dots n - L + 1$  do
6:        $j \leftarrow i + L - 1$ ,  $m[i, j] \leftarrow \infty$ 
7:       for  $k \leftarrow i \text{ to } j - 1$  do
8:          $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
9:         if  $q < m[i, j]$  then
10:           $m[i, j] \leftarrow q$ ,  $s[i, j] \leftarrow k$ 
11:        end if
12:      end for
13:    end for
14:  end for
15: end for
16: return  $m$  and  $s$ 
```

Runtime Analysis

- The array $s[i, j]$ is used to extract the actual sequence (see next).
- There are 3 nested loops and each can iterate at most n times, so the total running time is $\Theta(n^3)$.

Extracting Optimal Sequence

- Leave a split marker indicating where the best split is (i.e. the value of k leading to minimum values of $m[i, j]$).
- We maintain a parallel array $s[i, j]$ in which we store the value of k providing the optimal split.
- If $s[i, j] = k$, the best way to multiply the sub-chain $A_i \dots j$ is to first multiply the sub-chain $A_{i \dots k}$ and then the sub-chain $A_{k+1 \dots j}$, and finally multiply them together.
- Intuitively $s[i, j]$ tells us what multiplication to perform last.
- We only need to store $s[i, j]$ if we have at least 2 matrices where $j > i$.

Algorithm 4 $\text{Mult}(A,i,j)$

```
1: if  $i < j$  then  
2:    $k \leftarrow s[i,j]$   
3:    $X \leftarrow \text{Mult}(A, i, k)$  //  $X = A[i] \dots A[k]$   
4:    $Y = \text{Mult}(A, k + 1, j)$  //  $Y = A[k+1] \dots A[j]$   
5:   return  $X * Y$   
6: else  
7:   return  $A[i]$   
8: end if
```

Chain Multiplication – Example

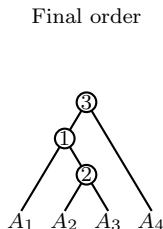
The initial set of dimensions are $\langle 5, 4, 6, 2, 7 \rangle$: we are multiplying A_1 (5×4) times A_2 (4×6) times A_3 (6×2) times A_4 (2×7). Optimal sequence is $(A_1(A_2A_3))A_4$.

Figure 1 shows a 4x4 matrix $m[i,j]$ representing the longest common subsequence (LCS) between two strings. The rows are indexed by i (0 to 4) and the columns by j (0 to 4). The matrix is filled with values representing the length of the LCS for substrings of the two strings. The first string is "A1A2A3A4" and the second string is "1234". The matrix is as follows:

	0	1	2	3	4
0	0	120	88	158	1
1	0	48	104	2	2
2	0	84	3	3	3
3	0	4	4	4	4

The values are computed using the recurrence relation: $m[i,j] = \max(m[i-1,j], m[i,j-1], m[i-1,j-1] + 1 \text{ if } A[i] == B[j])$.

			s[i,j]	
			j	
2	3	4		
1	1	3	1	
	2	3	2	i
		3	3	



Finding a Recursive Solution

- Figure out the "top-level" choice you have to make (e.g., where to split the list of matrices)
- List the options for that decision
- Each option should require smaller sub-problems to be solved
- Recursive function is the minimum (or max) over all the options

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j)$$

Steps in Dynamic Programming

- Characterize structure of an optimal solution.
- Define value of optimal solution recursively.
- Compute optimal solution values either **top-down** with caching or **bottom-up** in a table.
- Construct an optimal solution from computed values.