Dynamic Programming (Ch. 15)

Dynamic programming can provide a good solution for problems that take exponential time to solve by brute-force methods.

Typically applied to optimization problems, where there are many possible solutions, each solution has a particular value, and we wish to find the solution with an optimal (minimal or maximal) value.

For many of these problems, we must consider all subsets of a possibly very large set, so there are 2^n possible solutions -- too many to consider sequentially for large n.

Dynamic Programming

Divide-and-conquer algorithms find an optimal solution by partitioning a problem into independent subproblems, solving the subproblems recursively, and then combining the solutions to solve the original problem.

Dynamic programming is applicable when the sub-problems are not independent, i.e. when they share subsubproblems.

Dynamic Programming

Developed by Richard Bellman in the 1950s. Not a specific algorithm, but a technique (like divide-and-conquer).

This process takes advantage of the fact that subproblems have optimal solutions that lead to an overall optimal solution.

DP is often useful for problems with overlapping subproblems. These algorithms typically solve each subproblem once, record the result in a table, and use the information from the table to solve larger problems.

Computing the nth Fibonacci number is an example of a non-optimization problem to which dynamic programming can be applied.

$F(n) = F(n-1) + F(n-2)$ for $n >= 2$

$F(0) = 0$ and $F(1) = 1$.

Fibonacci Numbers

A straightforward, but inefficient algorithm to compute the nth Fibonacci number uses a top-down approach:

```
RFibonacci (n)
1. if n = 0 then return 0
2. else if n = 1 then return 1
3. else return RFibonacci (n-1) + RFibonacci (n-2)
```

This approach uses calls on the same number many times, leading to an exponential running time.

Fibonacci Numbers

A more efficient, bottom-up approach starts with 0 and works up to n, requiring only n values to be computed:

```
Fibonacci(n)
1. f[0] = 0
2. f[1] = 1
3. for  i = 2 … n
4.      f[i] = f[i-1] + f[i-2]
5. return f[n]
```

The technique of storing answers to smaller subproblems is called bottom-up programming.

Rod Cutting Problem

Problem: Find optimal way to cut a rod of length n

Given: rod of length n and a table of prices for rods of length 1..n

The table specifies that a rod of length i has a **price** p_i

Optimization problem is to find best set of cuts to get **maximum** price where
- Each cut is integer length
- Can use any number of cuts, from 0 to n – 1
- There is no cost for a cut

Rod Cutting Problem

Example rod lengths and values:

Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	17	17	20

Can cut rod in 2^{n-1} ways since each inch can have a cut or no cut.

Example: rod of length 4:

4	lengths
1,3	1 + 8 = 9
2,2	5 + 5 =10
3,1	8 + 1 = 9
1,1,2	1 + 1 + 5 = 7
1,2,1	1 + 5 + 1 = 7
2,1,1	5 + 1 + 1 = 7
1,1,1,1	1 + 1 + 1 + 1 = 4

Best price:
= two 2-inch pieces
= $p_2 + p_2$
= 5 + 5 = 10

Calculating Maximum Revenue

Compute the maximum revenue (r_i) for rods of length i

Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	17	17	20

Let's compute these values from the bottom up for i=4

- 1: 0 cuts = p_1
- 2: Compare p_2, p_1+p_1
- 3: Compare p_3, p_2+p_1, $p_1+p_1+p_1$
- 4: Compare p_4, p_1+p_3, p_3+p_1, p_2+p_2, $p_1+p_1+p_2$, $p_1+p_2+p_1$, $p_2+p_1+p_1$, $p_1+p_1+p_1+p_1$

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	2 + 2
5	13	2 + 3
6	17	6 (no cuts)
7	18	1 + 6 or 2 + 2 + 3
8	22	2 + 6

Cut-Rod

Recursive, top-down implementation:

```
Cut-Rod(p,n)
  1.  if n == 0
  2.      return 0
  3.  q = negative infinity
  4   for i = 1 to n
  5.      q = max(q, p[i] + CutRod(p, n – 1))
  6.  return q
```

T(n) =

Bottom-Up-Cut-Rod

Non-recursive, bottom-up implementation:

```
Bottom-Up-Cut-Rod(p,n):
1.  let r[0..n] be a new array
2.  r[0] = 0
3.  for j = 1 to n
4.      q = negative infinity
5.      for i = 1 to j
6.          if q <  p[ i ] + r[ j – i ]
7.              q = p[ i ] + r[ j – i ]
8.      r[ j ] = q
9.  return r[n]
```

r is the maximum revenue for each rod size.

T(n) =

Extended Bottom-Up-Cut-Rod

Non-recursive, bottom-up implementation that allows the enumeration of the max valued sequence of lengths:

```
Extended-Bottom-Up-Cut-Rod(p,n):
1.  let r[0..n] and s[0..n] be new arrays
2.  r[0] = 0
3.  for j = 1 to n
4.      q = negative infinity
5.      for i = 1 to j
6.          if q <  p[ i ] + r[ j – i ]
7.              q = p[ i ] + r[ j – i ]
8.              s[ j ] = i
9.      r[ j ] = q
10. return r[n] and s[n]
```

r is the maximum revenue for each rod size.

s is the optimal size of the first piece to cut off.

Print-Cut-Rod-Solution(p, n)

Takes a price table p and a rod size n and calls Extended-Bottom-Up-Cut-Rod

```
Print-Cut-Rod-Solution(p,n):
1.  (r, s) = Bottom-Up-Cut-Rod(p,n)
2.   while n > 0
3.      print s[n]
4.      n = n – s[n]
```

Matrix-Chain Product

If A is an $m \times n$ matrix and B is an $n \times p$ matrix, then

$$A \cdot B = C \quad \text{is an } m \times p \text{ matrix}$$

and the time needed to compute C is O(*mnp*).

- there are *mp* elements of C
- each element of C requires *n* scalar multiplications and *n-1* scalar additions

Matrix-Chain Multiplication Problem:

Given matrices $A_1, A_2, A_3, \ldots, A_n$, where the dimension of A_i is $p_{i-1} \times p_i$, determine the *minimum* number of multiplications needed to compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$. This involves finding the optimal way to *parenthesize* the matrices.

For more than 2 matrices, there exists more than one order of multi-plication.

Matrix-Chain Product

The running time of a brute-force solution (exhaustively checking all ways to parenthesize 2 matrices) is:

T(n) = 1 if n=1,

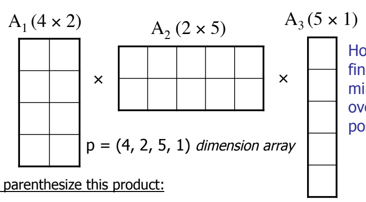
$$= \sum_{k=1}^{n-1} P(k)P(n-k) = \Omega(2^n) \quad \text{if } n \geq 2$$

Here, P(k) is the way to parenthesize first k matrices and P(n – k) is the way to parenthesize the rest.

Hopefully, we can do better using Dynamic Programming.

Matrix-Chain Product Example

The order of mult can make a difference



How do we find the minimum over all possibilities?

Two ways to parenthesize this product:

$(A_1 \cdot A_2) \cdot A_3$
$M_1 = A_1 \cdot A_2$: requires $4 \cdot 2 \cdot 5 = 40$ multiplications, M_1 is 4×5 matrix
$M_2 = M_1 \cdot A_3$: requires $4 \cdot 5 \cdot 1 = 20$ multiplications, M_2 is 4×1 matrix
–> total multiplications = 40 + 20 = 60

$A_1 \cdot (A_2 \cdot A_3)$
$M_1 = A_2 \cdot A_3$: requires $2 \cdot 5 \cdot 1 = 10$ multiplications, M_1 is 2×1 matrix
$M_2 = A_1 \cdot M_1$: requires $4 \cdot 2 \cdot 1 = 8$ multiplications, M_2 is 4×1 matrix
–> total multiplications = 10 + 8 = 18

Matrix-Chain Product

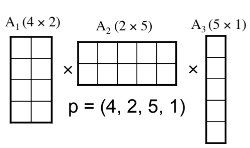
The optimal substructure of this problem can be given with the following argument:

Suppose an optimal way to parenthesize $A_i A_{i+1} \ldots A_j$ splits the product between A_k and A_{k+1}. Then the way the prefix subchain $A_i A_{i+1} \ldots A_k$ is parenthesized must be optimal. Why?

If there were a less costly way to parenthesize $A_i A_{i+1} \ldots A_k$, substituting that solution as the way to parenthesize $A_i A_{i+1} \ldots A_j$ gives a solution with lower cost, contradicting the assumption that the way the original group of matrices was parenthesized was optimal.

Therefore, the structure of the subproblems must be optimal.

Matrix-Chain Product – Recursive Solution



Let M[i,j] = min number of mults to compute $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$, where dimension of $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ is $p_{i-1} \times p_j$
M[i,i] = 0 for i = 1 to n, and
M[1,n] is the solution we want.

p is numbered from 0 to n and p_i is p[i].

M[i,j] can be determined as follows:

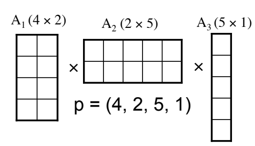
$$M[i,j] = \min(M[i,k] + M[k+1, j] + p_{i-1}\, p_k\, p_j), \text{ where } i \leq k < j,$$

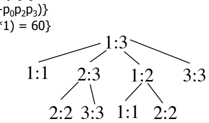
where M[i,j] equals the minimum cost for computing subproducts $A_{i \ldots k} A_{k+1 \ldots j}$ plus the cost of multiplying these two matrices together. Each matrix A_i is dimension p_{i-1} x p_i, so computing matrix product $A_{i \ldots k} A_{k+1 \ldots j}$ takes $p_{i-1}\, p_k\, p_j$ scalar multiplications.

Matrix-Chain Product – Recursive Solution

```
RMP(p, i, j)          // p is array of dimensions, initially i = 1, j = # of matrices
1. if i = j  return 0 // nothing need be done with a single matrix
2. M[i,j] = ∞
3. for k = i to j-1
4.     q = RMP(p, i, k) + RMP(p, k+1, j) + p_{i-1}p_k p_j
5.     if q < M[i,j] then M[i,j] = q
6. return M[i,j]
```

M[1,3] = min{(M[1,1]+M[2,3]+$p_0p_1p_3$= 0+$p_1p_2p_3$+$p_0p_1p_3$)
(M[1,2]+M[3,3]+$p_0p_2p_3$= $p_0p_1p_2$+0+$p_0p_2p_3$)}
= min {(2*5*1+4*2*1) = 18, (4*2*5+4*5*1) = 60}

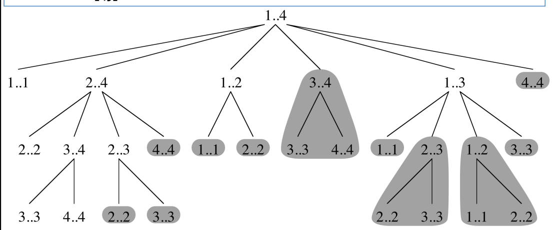




There is redundant computation because no intermediate results are stored.

Matrix-Chain Product – Recursive Solution

RMP(p, i, j) // p is array of dimensions, initially i = 1, j = # of matrices
1. if i = j return 0 // nothing need be done with a single matrix
2. M[i,j] = ∞
3. for k = i to j-1
4. q = RMP(p, i, k) + RMP(p, k+1, j) + $p_{i-1}p_kp_j$
5. if q < M[i,j] then M[i,j] = q
6. return M[i,j]



Matrix-Chain Product – Bottom-up solution

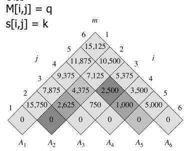
Matrix-Chain-Order (p) // p is array of dimensions
1. n = p.length – 1
2. let M[1…n, 1…n] and s[1…n-1,2…n] be new tables
3. for i = 1 to n
4. M[i,i] = 0 /** fill in main diagonal of M with 0s **/
5. for d = 2 to n /** d is chain length**/
6. for i = 1 to n – d + 1
7. j = i + d - 1
8. M[i, j] = ∞
9. for k = i to j-1
10. q = M[i, k]+ M[k+1,j] + $p_{i-1}p_kp_j$)
11. if q < M[i, j]
12. M[i, j] = q
13. s[i, j] = k
14. return M and s

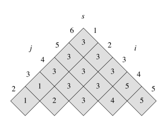
The M matrix holds the lowest number of multiplications for each sequence and s holds the split point for that sequence. Matrix s is used to print the optimal paren-thesization.

Matrix-Chain Product – Bottom-up solution

Matrix-Chain-Order (p) // p is array of dimensions
1. n = p.length – 1
2. let M[1…n, 1…n] and s[1…n-1,2…n] be new tables
3. for i = 1 to n
4. M[i,i] = 0 /** fill in main diagonal with 0s **/
5. for d = 2 to n /** d is chain length**/
6. for i = 1 to n – d + 1
7. j = i + d - 1
8. M[i,j] = ∞
9. for k = i to j-1
10. q = M[i,k]+ M[k+1,j] + $p_{i-1}p_kp_j$)
11. if q < M[i,j]
12. M[i,j] = q
13. s[i,j] = k
14. return M and s

p = [30, 35, 15, 5, 10, 20, 25]



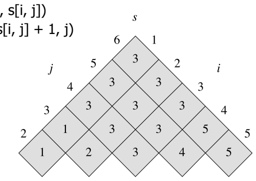


Matrix-Chain Product – Bottom-up solution

Input: s array, dimensions of M matrix
Output: side-effect printing of optimal parenthesization

Print-Optimal-Parens(s, i, j) // initially, i = 1 and j = n
1. if i == j
2. print "A"$_i$
3. else
4. print "("
5. Print-Optimal-Parens(s, i, s[i, j])
6. Print-Optimal-Parens(s, s[i, j] + 1, j)
7. print ")"

"((A1 (A2 A3))((A4 A5) A6))"



Matrix-Chain Product – Bottom-Up Solution

Complexity:
- O(n^3) time because of the nested for loops with each of d, i, and k taking on at most n-1 values.
- O(n^2) space for two n x n matrices M and s