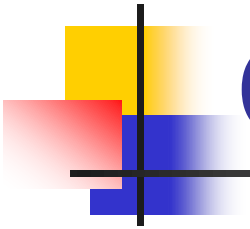
Introduction to Algorithms (Chapter 15: Dynamic Programming)

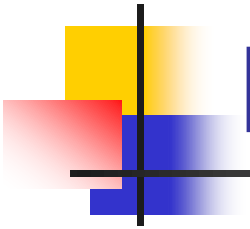
Kyuseok Shim

Electrical and Computer Engineering
Seoul National University



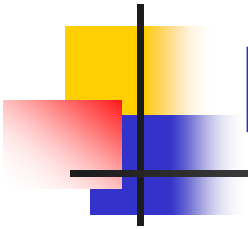
Outline

- This chapter covers an important technique called dynamic programming used in designing and analyzing efficient algorithms.



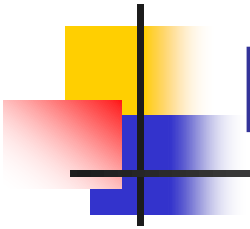
Divide-and-Conquer Method

- Partition the problem into disjoint sub-problems.
- Solve the sub-problems recursively
- Then, combine their solutions to solve the original problem.



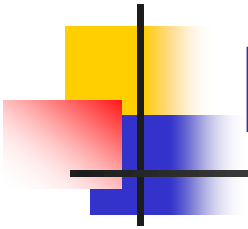
Dynamic Programming

- It is applicable when the sub-problems overlap - subproblems share subsubproblems.
- It solves every sub-problem just once and then saves its answer in a table, thereby avoiding re-computing the answer every time it solves each subsubproblem.



Dynamic Programming

- It is typically applied to optimization problems.
  - There can be many possible solutions.
  - Each solution has a value.
  - We wish to find a solution with the optimal value.
  - There can be several solutions achieving the optimal value.
  - We call such a solution an optimal solution to the problem, as opposed to the optimal solution.



Dynamic Programming

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution in a bottom-up fashion.
- Construct an optimal solution from computed information.

Rod Cutting

Rod Cutting Problem

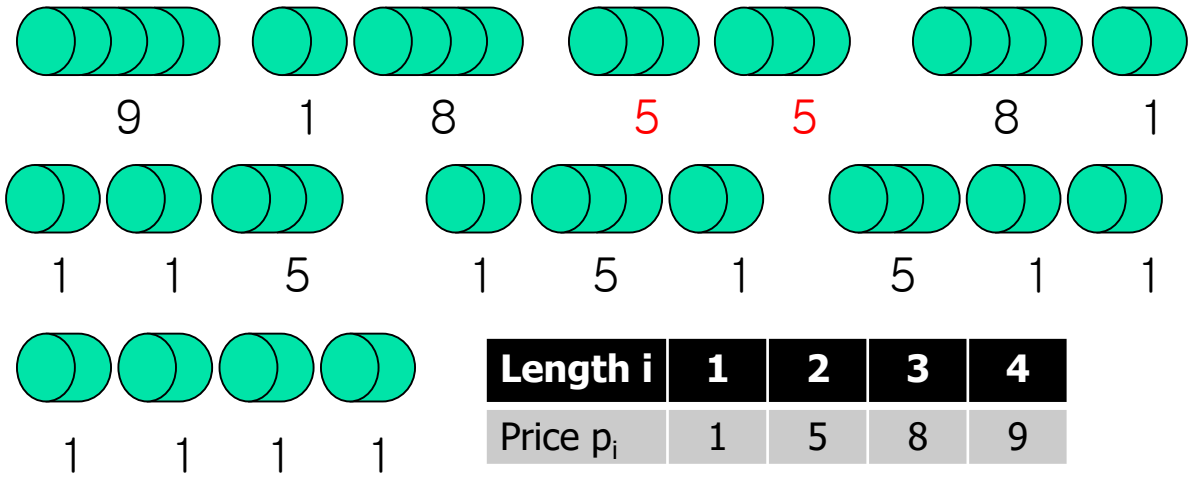
- Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells.
- Each cut is free.
- The management of Serling Enterprises wants to know the best way to cut up the rods.
- We know that the price p_i in dollars that Serling Enterprises charges for a rod of length i inches.
- Rod lengths are always an integral number of inches.

Rod Cutting Problem

- Given a rod of length n inches and a table of prices p_i for i=1,2,…,n.
- Determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.
- Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

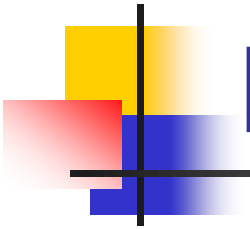
Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

An Example of n = 4



Length i	1	2	3	4
Price p_i	1	5	8	9

8 possible ways of cutting up a rod of length 4

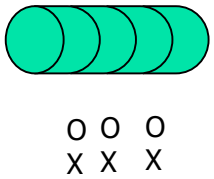


Rod Cutting Problem

- How many different ways of cutting up a rod of length n?

Rod Cutting Problem

- How many different ways of cutting up a rod of length n?
  - There are 2^{n-1} different ways since we have an independent option of cutting, or not cutting, at distance i inches from the left end, for i=1,…,n-1



8 possible ways of cutting up a rod of length 4

Rod Cutting Problem

- We denote a decomposition into pieces using ordinary additive notation, so that 7 = 2 + 2 + 3 indicates that a rod of length 7 is cut into three pieces - two of length 2 and one of length 3.
- If an optimal solution cuts the rod into k pieces, for some $1 \le k \le n$, then an optimal decomposition

$$n = i_1 + i_2 + \ldots + i_k$$

of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$

provide maximum corresponding revenue

$$r_n = p_{i1} + p_{i2} + \ldots + p_{ik}.$$

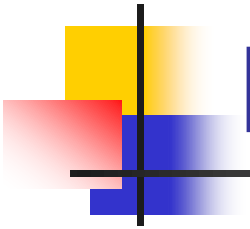
An Example of Optimal Solutions

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

- r_1=1 from solution 1=1 (no cuts)
- r_2=5 from solution 2=2 (no cuts)
- r_3=8 from solution 3=3 (no cuts)
- r_4=10 from solution 4=2+2
- r_5=13 from solution 5=2+3
- r_6=17 from solution 6=6 (no cuts)
- r_7=18 from solution 7=1+6 or 2+2+3
- r_8=22 from solution 8=2+6
- r_9=25 from solution 9=3+6

Rod Cutting

- We can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods as
- $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) = \max(p_n, \max_{1 \leq i < n}(r_i + r_{n-i}))$
  - P_n: corresponds to making no cuts at all and selling the rod of length n as is
  - The other n-1 arguments to max correspond to the maximum revenue obtained by making an initial cut of the rod into two pieces of size i and n-i , and then optimally cutting up those pieces further, obtaining revenues r_n and r_{n-1} from those two pieces.
- Since we don't know ahead of time which value of i optimizes revenue, we have to consider all possible values for i and pick the one that maximizes revenue.
- We also have the option of picking no i at all if we can obtain more revenue by selling the rod uncut.



Rod Cutting

- To solve the original problem of size n, we solve smaller problems of the same type, but of smaller sizes.
- Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem.
- The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces.
- We say that the rod-cutting problem exhibits **optimal substructure**
  - Optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

Recurrence for Time Complexity?

- $r_n = \max(p_n, \max_{1 \le i < n}(r_i + r_{n-i}))$

Recurrence for Time Complexity?

- $r_n = \max(p_n, \max_{1 \le i < n}(r_i + r_{n-i}))$

$$T(n) = 1 + \sum_{j=1}^{n-1} \big(T(j) + T(n-j)\big)$$

Recurrence for Time Complexity?

- $r_n=\max(p_n,\max_{1\le i<n}(r_i+r_{n-i}))$

$$T(n) = 1 + \sum_{j=1}^{n-1} \big(T(j) + T(n-j)\big)$$

$$T(n-1) = 1 + \sum_{j=1}^{n-2} \big(T(j) + T(n-1-j)\big)$$

$$T(n) - T(n-1) = \sum_{j=1}^{n-1} \big(T(j) + T(n-j)\big) - \sum_{j=1}^{n-2} \big(T(j) + T(n-1-j)\big) = 2T(n-1)$$

$$T(n) = 3T(n-1)$$

$$T(n) = T(0) \cdot 3^n = 3^n$$

Rod Cutting

- A simpler way to arrange a recursive structure for the rod cutting problem is to view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length n - i.
- Only the remainder may be further divided.
- We can couch the solution with no cuts at all as saying that the first piece has size i= n with revenue p_n and the remainder has size 0 with corresponding revenue $r_0=0$.
- An optimal solution embodies the solution to only one related subproblem rather than two
  - $r_n=\max_{1\le i\le n}(p_i+r_{n-i})$ (first piece : size i and revenue p_i)

Recursive Top-down Implementation

```
CUT-ROD(p,n)
1.  if n==0
2.      return 0
3.  q=-∞
4.  for i=1 to n
5.      q=max(q, p[i]+CUT-ROD(p,n-i))
6.  return q
```

Recurrence for Time Complexity?

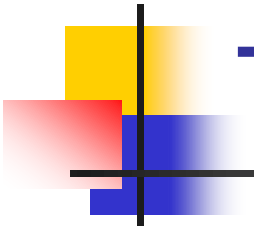
```
CUT-ROD(p,n)
1.  if n==0
2.      return 0
3.  q=-∞
4.  for i=1 to n
5.      q=max(q, p[i]+CUT-ROD(p,n-i))
6.  return q
```

Recurrence for Time Complexity?

CUT-ROD(p,n)

1. **if** n==0
2. **return** 0
3. q=-∞
4. **for** i=1 **to** n
5. q=max(q, p[i]+CUT-ROD(p,n-i))
6. **return** q

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$



T(n)?

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

T(n)?

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

$$T(n-1) = 1 + \sum_{j=0}^{n-2} T(j)$$

$$T(n) - T(n-1) = \sum_{j=0}^{n-1} T(j) - \sum_{j=0}^{n-2} T(j) = T(n-1)$$

$$T(n) = 2T(n-1)$$

$$T(n) = T(0) \cdot 2^n = 2^n$$

T(n)?

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

$$T(n) = 2^n$$

- This exponential running time is not so surprising.
  - CUT-ROD explicitly considers all the 2^{n-1} possible ways of cutting up a rod of length n.
  - The tree of recursive calls has 2^{n-1} leaves, one for each possible way of cutting up the rod.

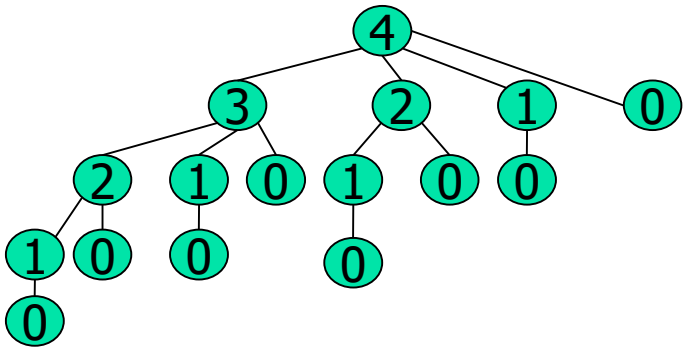
Inefficient, Why?

```
CUT-ROD(p,n)
1.  if n==0
2.      return 0
3.  q=-∞
4.  for i=1 to n
5.      q=max(q, p[i]+CUT-ROD(p,n-i))
6.  return q
```

- The problem is that CUT-ROD calls itself recursively over and over again with the same parameter values.
- It solves the same subproblems repeatedly.

What Happens for n=4

- The naïve recursive solution is inefficient - solve the same subproblems repeatedly.
- The recursion tree showing recursive calls resulting from a call CUT-ROD(p,n) for n=4.
- A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n.
- In general, the recursion tree 2^n nodes and 2^{n-1} leaves.



Dynamic Programming for Optimal Rod Cutting

- Since a naive recursive solution solves the same subproblems repeatedly, we solve each subproblem only *once* by saving its solution.
- When we refer to this subproblem's solution again later, we just look it up, rather than recompute it.
- Dynamic programming thus uses additional memory to save computation time - time-memory trade-off.
- The savings may be dramatic - an exponential-time solution may be transformed into a polynomial-time solution.
- A dynamic-programming approach runs in polynomial time if the number of distinct subproblems involved is polynomial in the input size and we can solve each such subproblem in polynomial time.

Implementing a Dynamic Programming Approach

- Top-down method with memoization
  - Write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table).
  - First checks whether it has previously solved this subproblem.
  - If so, return the saved value.
  - If not, computes the value in the usual manner and save the result - memoized - it "remembers" what results it has computed previously.

Implementing a Dynamic Programming Approach

- Bottom-up method
  - Utilize some natural notion of the "size" of a subproblem.
  - Sort the subproblems by size and solve them in size order - smallest first.
  - When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions.
  - Solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

Implementing a Dynamic Programming Approach

- Both top-down and bottom-up approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems.
- The bottom-up approach often has much better constant factors, since it has less overhead for procedure calls.

Top-down Version

```
MEMOIZED-CUT-ROD(p,n)
1.  let r[0…n] be a new array
2.  for i=0 to n
3.      r[i]=-∞
4.  return MEMOIZED-CUT-ROD-AUX(p,n,r)

MEMOIZED-CUT-ROD-AUX(p,n,r)
1.  if r[n]≥0
2.      return r[n]
3.  if n==0
4.      q=0
5.  else
6.      q=-∞
7.      for i=1 to n
8.          q=max(q,p[i]+MEMOIZED-CUT-ROD-AUX (p,n-i,r))
9.  r[n]=q
10. return q
```

Bottom-up Version

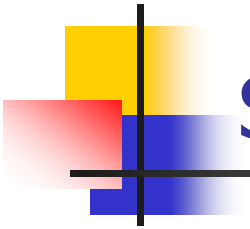
```
BOTTOM-UP-CUT-ROD(p,n)
1.  let r[0…n] be a new array
2.  r[0]=0
3.  for j=1 to n
4.      q=-∞
5.      for i=1 to j
6.          q=max(q,p[i]+r[j-i])
7.      r[j]=q
8.  return r[n]
```

- $T(n)=\Theta(n^2)$ - A doubly-nested loop structure
- The number of iterations of its inner **for** loop, in lines 5–6, forms an arithmetic series.

Top-down Version

```
MEMOIZED-CUT-ROD-AUX(p,n,r)
1.  if r[n]≥0
2.      return r[n]
3.  if n==0
4.      q=0
5.  else
6.      q=-∞
7.      for i=1 to n
8.          q=max(q,p[i]+MEMOIZED-CUT-ROD-AUX (p,n-i,r))
9.  r[n]=q
10. return q
```

- $T(n)=\Theta(n^2)$
- Since a recursive call to a previously solved subproblem returns immediately, we solve each subproblem with sizes 0, 1,…,n just once.
- Thus, the number of iterations of this **for** loop, over all recursive calls of MEMOIZED-CUT-ROD, forms an arithmetic series.

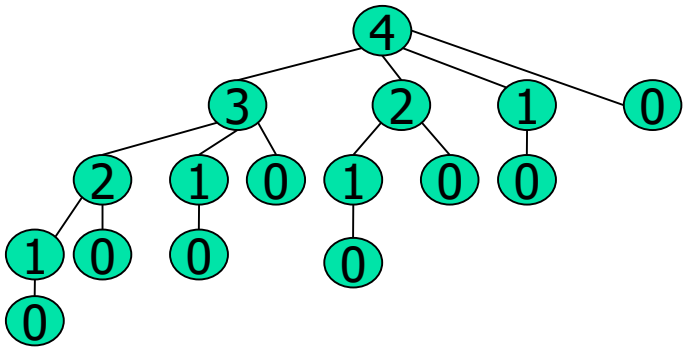


Subproblem Graph

- The subproblem graph illustrates the set of subproblems involved and how subproblems depend on one another.
- It is a directed graph, containing one vertex for each distinct subproblem.
- The subproblem graph has a directed edge from the vertex for subproblem x to the vertex for subproblem y if determining an optimal solution for subproblem x involves directly considering an optimal solution for subproblem y.
- We can think of the subproblem graph as a "collapsed" version of the recursion tree for the top-down recursive method, in which we coalesce all nodes for the same subproblem into a single vertex and direct all edges from parent to child.

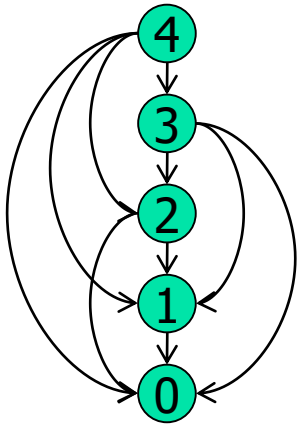
Recursion Tree for Top-down Recursive Method

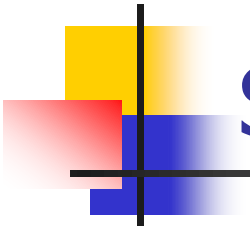
- The recursion tree showing recursive calls resulting from a call CUT-ROD(p,n) for n=4.
- A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n.
- In general, the recursion tree 2^n nodes and 2^{n-1} leaves.



Subproblem Graph

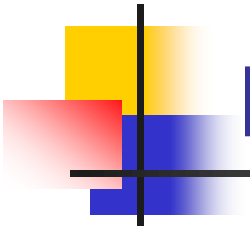
- The subproblem graph of top-down version of dynamic programming algorithm for the rod-cutting problem with n = 4.





Subproblem Graph

- The size of the subproblem graph determines the running time of the dynamic programming algorithm.
- Since we solve each subproblem just once, the running time is the sum of the times needed to solve each subproblem.
- The time to compute the solution to a subproblem is proportional to the number of outgoing edges of the corresponding vertex, and the number of subproblems is equal to the number of vertices.
- Thus, the running time of dynamic programming is linear in the number of vertices and edges.



Reconstructing a Solution

- Our dynamic-programming solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual list of piece sizes.
- We can extend the dynamic-programming approach to record
  - not only the optimal value computed for each subproblem
  - but also a choice that led to the optimal value
- With this information, we can readily print an optimal solution.

Reconstructing a Solution

```
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=-∞
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 and s
```

- Update s[j] in line 8 to hold the optimal size i of the first piece to cut off when solving a subproblem of size j.

Reconstructing a Solution

```
PRINT-CUT-ROD-SOLUTION(p,n)
1.   (r, s) =  EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
2.   while n > 0
3.       print s[n]
4.       n=n - s[n]
```

- The procedure takes a price table p and a rod size n.
- It calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the array s[1..n] of optimal first-piece sizes and then prints out the complete list of piece sizes in an optimal decomposition of a rod of length n.

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

i	0	1	2	3	4	5	6	7	8	9
r[i]										
s[i]										

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

i	0	1	2	3	4	5	6	7	8	9
r[i]	0									
s[i]										

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 1

q = -∞

i	0	1	2	3	4	5	6	7	8	9
r[i]	0									
s[i]										

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 1
i = 1
q = 1

i	0	1	2	3	4	5	6	7	8	9
r[i]	0									
s[i]		1								

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 1
i = 2
q = 1

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1								
s[i]		1								

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 2

q = -∞

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1								
s[i]		1								

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 2
i = 1
q = 2

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1								
s[i]		1	1							

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 2
i = 2
q = 5

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1								
s[i]		1	2							

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 2
i = 3
q = 5

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5							
s[i]		1	2							

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 3

q = -∞

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5							
s[i]		1	2							

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 3
i = 1
q = 6

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5							
s[i]		1	2	1						

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 3
i = 2
q = 6

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5							
s[i]		1	2	1						

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 3
i = 3
q = 8

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5							
s[i]		1	2	3						

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 3
i = 4
q = 8

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8						
s[i]		1	2	3						

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 4

q = -∞

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8						
s[i]		1	2	3						

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 4
i = 1
q = 9

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8						
s[i]		1	2	3	1					

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 4
i = 2
q = 10

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8						
s[i]		1	2	3	2					

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 4
i = 3
q = 10

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8						
s[i]		1	2	3	2					

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 4
i = 4
q = 10

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8						
s[i]		1	2	3	2					

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 4
i = 5
q = 10

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10					
s[i]		1	2	3	2					

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 5
i = 6
q = 13

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13				
s[i]		1	2	3	2	2				

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 6
i = 7
q = 17

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17			
s[i]		1	2	3	2	2	6			

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 7
i = 8
q = 18

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18		
s[i]		1	2	3	2	2	6	1		

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 8
i = 9
q = 22

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	
s[i]		1	2	3	2	2	6	1	2	

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 9
i = 10
q = 25

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	25
s[i]		1	2	3	2	2	6	1	2	3

Reconstructing a Solution

```
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=-∞
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 and s
```

Length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	24

j = 10

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	25
s[i]		1	2	3	2	2	6	1	2	3

Reconstructing a Solution

```
PRINT-CUT-ROD-SOLUTION(p,n)
1.    (r, s) =  EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
2.    while n > 0
3.        print s[n]
4.        n = n - s[n]
```

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	25
s[i]	0	1	2	3	2	2	6	1	2	3

- PRINT-CUT-ROD-SOLUTION(p,9) - print 3 and 6

Reconstructing a Solution

```
PRINT-CUT-ROD-SOLUTION(p,n)
1.    (r, s) =  EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
2.    while n > 0
3.        print s[n]
4.        n = n - s[n]
```

- PRINT-CUT-ROD-SOLUTION(p,9)

Reconstructing a Solution

```
PRINT-CUT-ROD-SOLUTION(p,n)
1.   (r, s) =  EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
2.   while n > 0
3.       print s[n]
4.       n = n - s[n]
```

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	25
s[i]	0	1	2	3	2	2	6	1	2	3

- PRINT-CUT-ROD-SOLUTION(p,9)

Reconstructing a Solution

```
PRINT-CUT-ROD-SOLUTION(p,n)
1.    (r, s) =  EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
2.    while n > 0
3.        print s[n]
4.        n = n - s[n]
```

n = 9

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	25
s[i]	0	1	2	3	2	2	6	1	2	3

- PRINT-CUT-ROD-SOLUTION(p,9)

Reconstructing a Solution

```
PRINT-CUT-ROD-SOLUTION(p,n)
1.    (r, s) =  EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
2.    while n > 0
3.        print s[n]
4.        n = n - s[n]
```

n = 9

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	25
s[i]	0	1	2	3	2	2	6	1	2	3

- PRINT-CUT-ROD-SOLUTION(p,9)

Reconstructing a Solution

PRINT-CUT-ROD-SOLUTION(p,n)

1. (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
2. **while** n > 0
3. print s[n]
4. n = n - s[n]

n = 9

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	25
s[i]	0	1	2	3	2	2	6	1	2	3

- PRINT-CUT-ROD-SOLUTION(p,9) - print 3

Reconstructing a Solution

PRINT-CUT-ROD-SOLUTION(p,n)

1. (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
2. **while** n > 0
3. print s[n]
4. n = n - s[n]

n = 9 – 3 = 6

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	25
s[i]	0	1	2	3	2	2	6	1	2	3

- PRINT-CUT-ROD-SOLUTION(p,9) - print 3

Reconstructing a Solution

```
PRINT-CUT-ROD-SOLUTION(p,n)
1.    (r, s) =  EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
2.    while n > 0
3.        print s[n]
4.        n = n - s[n]
```

n = 6

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	25
s[i]	0	1	2	3	2	2	6	1	2	3

- PRINT-CUT-ROD-SOLUTION(p,9) - print 3

Reconstructing a Solution

PRINT-CUT-ROD-SOLUTION(p,n)

1. (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
2. **while** n > 0
3. print s[n]
4. n = n - s[n]

n = 6

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	25
s[i]	0	1	2	3	2	2	6	1	2	3

- PRINT-CUT-ROD-SOLUTION(p,9) - print 3, print 6

Reconstructing a Solution

PRINT-CUT-ROD-SOLUTION(p,n)

1. (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
2. **while** n > 0
3. print s[n]
4. n = n - s[n]

n = 6 – 6 = 0

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	25
s[i]	0	1	2	3	2	2	6	1	2	3

- PRINT-CUT-ROD-SOLUTION(p,9) - print 3, print 6

Reconstructing a Solution

```
PRINT-CUT-ROD-SOLUTION(p,n)
1.    (r, s) =  EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
2.    while n > 0                                  n = 0
3.        print s[n]
4.        n = n - s[n]
```

i	0	1	2	3	4	5	6	7	8	9
r[i]	0	1	5	8	10	13	17	18	22	25
s[i]	0	1	2	3	2	2	6	1	2	3

- PRINT-CUT-ROD-SOLUTION(p,9) - print 3, print 6

Matrix-chain Multiplication

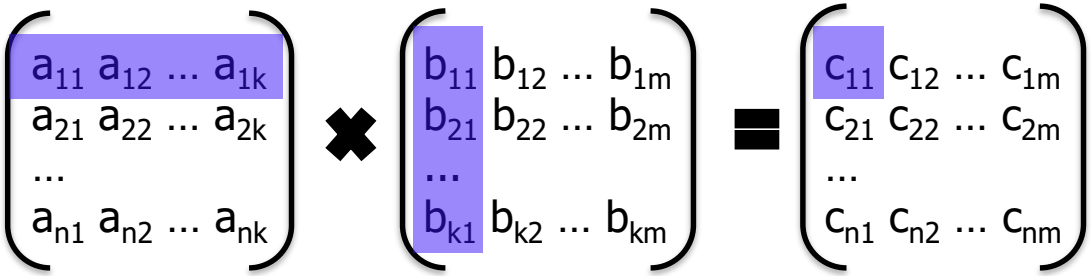
Matrix-chain Multiplication

- A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parenthesis.
- For a chain of matrices $\langle A_1, A_2, A_3, A_4 \rangle$, we can fully parenthesize the product in 5 distinct ways:
  - $(A_1(A_2(A_3A_4)))$
  - $(A_1((A_2A_3)A_4))$
  - $((A_1A_2)(A_3A_4))$
  - $((A_1(A_2A_3))A_4)$
  - $(((A_1A_2)A_3)A_4)$

Matrix Multiplication

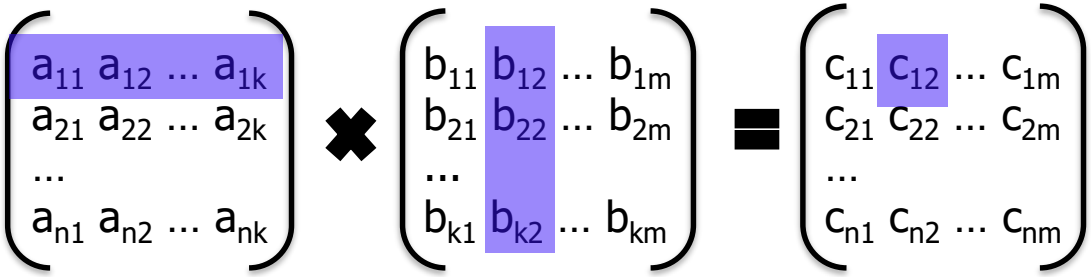
$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1k} \\ a_{21} & a_{22} & \dots & a_{2k} \\ \dots & & & \\ a_{n1} & a_{n2} & \dots & a_{nk} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \dots & & & \\ b_{k1} & b_{k2} & \dots & b_{km} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1m} \\ c_{21} & c_{22} & \dots & c_{2m} \\ \dots & & & \\ c_{n1} & c_{n2} & \dots & c_{nm} \end{pmatrix}$$

Matrix Multiplication



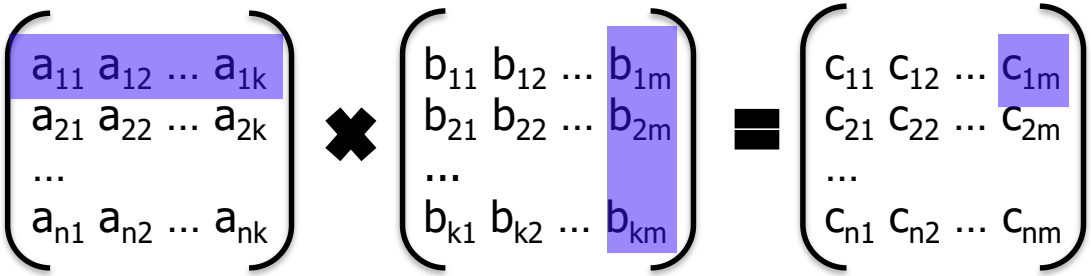
$$c_{11}=a_{11}*b_{11}+a_{12}*b_{21}+\ldots+a_{1k}*b_{k1}$$

Matrix Multiplication



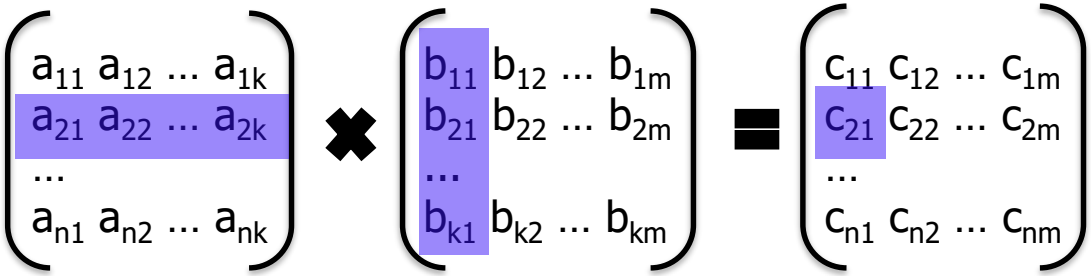
$$c_{12}=a_{11}*b_{12}+a_{12}*b_{22}+\ldots+a_{1k}*b_{k2}$$

Matrix Multiplication



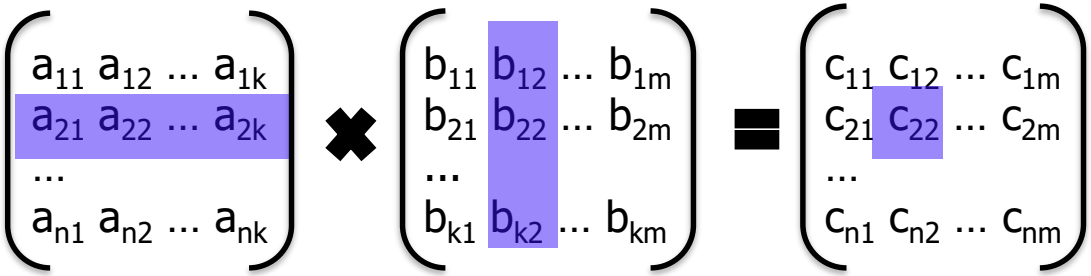
$$c_{1m}=a_{11}*b_{1m}+a_{12}*b_{2m}+\ldots+a_{1k}*b_{km}$$

Matrix Multiplication



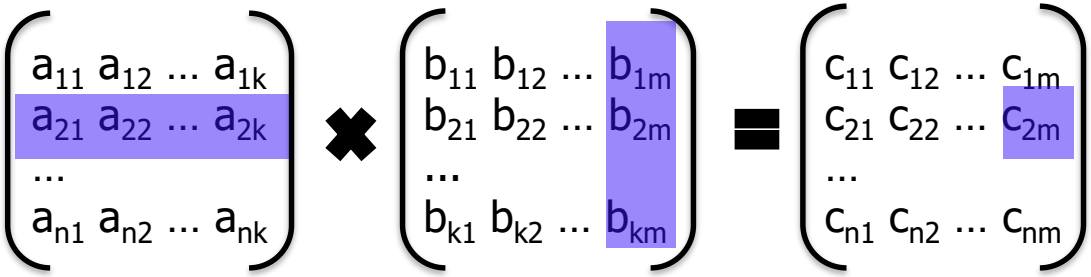
$$c_{21}=a_{21}*b_{11}+a_{22}*b_{21}+\ldots+a_{2k}*b_{k1}$$

Matrix Multiplication



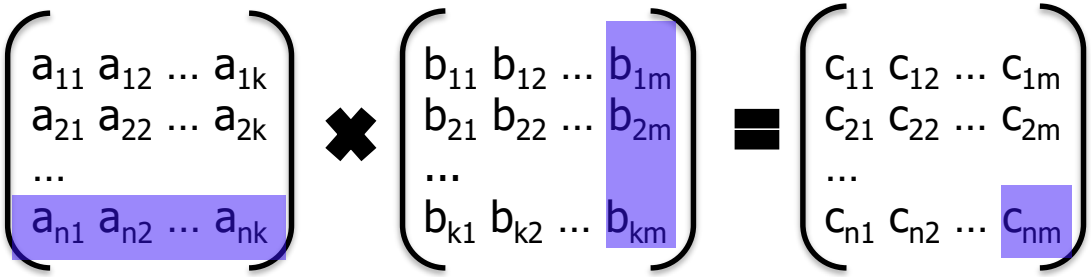
$$c_{22}=a_{21}*b_{12}+a_{22}*b_{22}+\ldots+a_{2k}*b_{k2}$$

Matrix Multiplication



$$c_{2m}=a_{21}*b_{1m}+a_{22}*b_{2m}+\ldots+a_{2k}*b_{km}$$

Matrix Multiplication



$$c_{nm}=a_{n1}*b_{1m}+a_{n2}*b_{2m}+\ldots+a_{nk}*b_{km}$$

Matrix Multiplication

MATRIX-MULTIPLY(A, B)

1. **if** A.column $\neq$ B.rows
2. **error** "incompatable dimensions"
3. **else** let C be a new A.rows X B.columns
4. **for** i=1 **to** A.rows
5. **for** j=1 **to** B.columns
6. $c_{ij}=0$
7. **for** k=1 **to** A.columns
8. $c_{ij}=c_{ij}+a_{ik}\cdot b_{kj}$
9. **return** C

- We can multiply two matrices A and B only if they are compatible: the number of columns of A must equal the number of rows of B.
- If A is a p×q matrix and B is a q×r matrix, the resulting matrix C is a p×r matrix.
- The time to compute C is dominated by the number of scalar multiplications in line 8, which is pqr.

An Example

- Consider the problem of a chain $<A_1, A_2, A_3>$ of three matrices
- A_1: 10×100, A_2: 100×5, A_3: 5×50

i. $((A_1A_2)A_3)$ **= 5000+2500 = 7500**

$(A_1 \times A_2)$: A1: 10 × 100 ⇒ 10 × 5 matrix

A2: 100 × 5

10 × 100 × 5 = 5000 scalar multiplications

$((A_1 \times A_2) \times A_3)$: A1 × A2 : 10 × 5 ⇒ 10 × 50 matrix

A3: 5 × 50

10 × 5 × 50 = 2500 scalar multiplications

An Example

- Consider the problem of a chain $<A_1, A_2, A_3>$ of three matrices
- A_1: 10×100, A_2: 100×5, A_3: 5×50

ii. $(A_1(A_2A_3))$ = 50000+25000 = 75000

$(A_2 \times A_3)$: A1: 100 × 5 ⇒ 100 × 50 matrix

A2: 5 × 50

100 × 5 × 50 = 25000 scalar multiplications

$(A_1 \times (A_2 \times A_3))$: A1 :10 ×100 ⇒ 10 × 50 matrix

A2 × A3 : 100 × 50

10 × 100 × 50 = 50000 scalar multiplications

An Example

- Consider the problem of a chain $<A_1, A_2, A_3>$ of three matrices
- A_1: 10×100, A_2: 100×5, A_3: 5×50

i. $((A_1A_2)A_3)$ = 5000+2500 = 7500

ii. $(A_1(A_2A_3))$ = 50000+25000 = 75000

Matrix-chain Multiplication Problem

- Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where A_i has dimension $p_{i-1} \times p_i$
- Fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

Counting the Number of Parenthesizations

- Denote the number of alternative parenthesizations of a sequence of n matrices by P(n)
- A fully parenthesized matrix product is the product of two fully parenthesized matrix subproblems.
  - $(A_1(A_2(A_3A_4)))$
  - $(A_1((A_2A_3)A_4))$
  - $((A_1A_2)(A_3A_4))$
  - $((A_1(A_2A_3))A_4)$
  - $(((A_1A_2)A_3)A_4)$
- The split between the two subproducts may occur between the k-th an (k+1)-st matrices for any k.

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

  - The solution to the recurrence is $\Omega(4^n/n^{3/2})$

Applying Dynamic Programming

- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution in a bottom-up fashion
- Construct an optimal solution from computed information

Step 1: Optimal Structure

- Find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems.
- $A_{i..j}$: the matrix resulting from evaluating the product $A_iA_{i+1}...A_j$
- We must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$.
- That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$.
- The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing the matrix $A_{k+1..j}$, plus the cost of multiplying them together.

Step 1: Optimal Structure

- Suppose that to optimally parenthesize $A_iA_{i+1}...A_j$, we split the product between A_k and A_{k+1}.
- Then, the way we parenthesize the prefix subchain $A_iA_{i+1}...A_k$ within this optimal parenthesization of $A_iA_{i+1}...A_j$, must be an optimal parenthesization of $A_iA_{i+1}...A_k$. Why? If there were a less costly way to parenthesize $A_iA_{i+1}...A_k$ to produce another way parenthesize $A_iA_{i+1}...A_j$ whose cost was lower than the optimum: A contradiction!
- A similar observation holds for how we parenthesize the subchain $A_{k+1}...A_j$ in the optimal parenthesization of $A_iA_{i+1}...A_j$.

Step 2: A Recursive Solution

- Define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems.
- We pick as our subproblems the problems of determining the minimum cost of parenthesizing c for $1 \leq i \leq j \leq n$.
- Let m[i, j] be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$.
- For the full problem, the lowest-cost way to compute $A_{1..n}$ would thus be m[1,n].
- Suppose that we split the product between A_k and A_{k+1}.
- Computing the matrix product $A_{i..k} A_{k+1..j}$ takes $p_{i-1}p_kp_j$ scalar multiplications.
- We can define m[i, j] recursively as follows.
  - If $i = j$, $m[i, j] = 0$ for $1 \leq i = j \leq n$.
  - When $i < j$, $m[i,j] = m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$.

Step 2: A Recursive Solution

- We can define m[i, j] recursively as follows.
  - If i = j , m[i, j] =0 for $1 \leq i = j \leq n$.
  - When i < j, $m[i,j] = m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$.
- This recursive equation assumes that we know the value of k, which we do not.
- There are only j - i possible values for k, however, namely k = i, i+1…,j-1.
- Since the optimal parenthesization must use one of these values for k, we need only check them all to find the best.
- Thus, our recursive definition for the minimum cost of parenthesizing the product $A_iA_{i+1}…A_j$ becomes

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

Step 2: A Recursive Solution

- Our recursive definition for the minimum cost of parenthesizing the product $A_iA_{i+1}...A_j$ is

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

- The m[i, j] values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution.
- Thus, we define s[i, j] to be a value of k at which we split the product $A_iA_{i+1}...A_j$ in an optimal parenthesization.

Step 3: Computing the Optimal Cost

```
MATRIX-CHAIN-ORDER(p) // p = <p0, p1, …, pn>
1.    n= p.length-1
2.    for i=1 to n
3.       m[i,i]=0
4.    for l=2 to n              // l is the chain length
5.       for i=1 to n-l+1
6.          j=i+l-1
7.          m[i,j]=∞
8.          for k=i to j-1
9.             q=m[i,k]+m[k+1,j]+p_{i-1}p_k p_j
10.            if q < m[i,j]
11.               m[i,j] = q
12.               s[i,j] = k
13.   return m and s
```

Step 3: Computing the Optimal Cost

```
MATRIX-CHAIN-ORDER(p) // p = <p0, p1, …, pn>
1.    n= p.length-1
2.    for i=1 to n
3.       m[i,i]=0
4.    for l=2 to n              // l is the chain length
5.       for i=1 to n-l+1
6.          j=i+l-1
7.          m[i,j]=∞
8.          for k=i to j-1
9.             q=m[i,k]+m[k+1,j]+p_{i-1}p_k p_j
10.            if q < m[i,j]
11.               m[i,j] = q
12.               s[i,j] = k
13.   return m and s
```

	i=1	i=2	…	i=n-3	i=n-2	i=n-1
l = 2	m[1,2]	m[2,3]	…	m[n-3,n-2]	m[n-2,n-1]	m[n-1,n]

Step 3: Computing the Optimal Cost

```
MATRIX-CHAIN-ORDER(p) // p = <p0, p1, …, pn>
1.    n= p.length-1
2.    for i=1 to n
3.       m[i,i]=0
4.    for l=2 to n              // l is the chain length
5.       for i=1 to n-l+1
6.          j=i+l-1
7.          m[i,j]=∞
8.          for k=i to j-1
9.             q=m[i,k]+m[k+1,j]+pi-1pkpj
10.            if q < m[i,j]
11.               m[i,j] = q
12.               s[i,j] = k
13.   return m and s
```

i=1 to n-l+1 (n-2+1)

	i=1	i=2	…	i=n-3	i=n-2	i=n-1
l = 2	m[1,2]	m[2,3]	…	m[n-3,n-2]	m[n-2,n-1]	m[n-1,n]

Step 3: Computing the Optimal Cost

```
MATRIX-CHAIN-ORDER(p) // p = <p0, p1, …, pn>
1.    n= p.length-1
2.    for i=1 to n
3.       m[i,i]=0
4.    for l=2 to n              // l is the chain length
5.       for i=1 to n-l+1
6.          j=i+l-1
7.          m[i,j]=∞
8.          for k=i to j-1
9.             q=m[i,k]+m[k+1,j]+p_{i-1}p_k p_j
10.            if q < m[i,j]
11.               m[i,j] = q
12.               s[i,j] = k
13.   return m and s
```

	i=1	i=2	…	i=n-3	i=n-2	i=n-1
l = 2	m[1,2]	m[2,3]	…	m[n-3,n-2]	m[n-2,n-1]	m[n-1,n]
l = 3	m[1,3]	m[2,4]	…	m[n-3,n-1]	m[n-2,n]	X

Step 3: Computing the Optimal Cost

```
MATRIX-CHAIN-ORDER(p) // p = <p0, p1, …, pn>
1.    n= p.length-1
2.    for i=1 to n
3.       m[i,i]=0
4.    for l=2 to n              // l is the chain length
5.       for i=1 to n-l+1
6.          j=i+l-1
7.          m[i,j]=∞
8.          for k=i to j-1
9.             q=m[i,k]+m[k+1,j]+pi-1pkpj
10.            if q < m[i,j]
11.               m[i,j] = q
12.               s[i,j] = k
13.   return m and s
```

i=1 to n-l+1 (n-3+1)

	i=1	i=2	…	i=n-3	i=n-2	i=n-1
l = 2	m[1,2]	m[2,3]	…	m[n-3,n-2]	m[n-2,n-1]	m[n-1,n]
l = 3	m[1,3]	m[2,4]	…	m[n-3,n-1]	m[n-2,n]	X

Step 3: Computing the Optimal Cost

```
MATRIX-CHAIN-ORDER(p) // p = <p0, p1, …, pn>
1.    n= p.length-1
2.    for i=1 to n
3.       m[i,i]=0
4.    for l=2 to n              // l is the chain length
5.       for i=1 to n-l+1
6.          j=i+l-1
7.          m[i,j]=∞
8.          for k=i to j-1
9.             q=m[i,k]+m[k+1,j]+p_{i-1}p_k p_j
10.            if q < m[i,j]
11.               m[i,j] = q
12.               s[i,j] = k
13.   return m and s
```

	i=1	i=2	…	i=n-3	i=n-2	i=n-1
l = 2	m[1,2]	m[2,3]	…	m[n-3,n-2]	m[n-2,n-1]	m[n-1,n]
l = 3	m[1,3]	m[2,4]	…	m[n-3,n-1]	m[n-2,n]	X
l = 4	m[1,4]	m[2,5]	…	m[n-3,n]	X	X

Step 3: Computing the Optimal Cost

```
MATRIX-CHAIN-ORDER(p) // p = <p0, p1, …, pn>
1.    n= p.length-1
2.    for i=1 to n
3.       m[i,i]=0
4.    for l=2 to n              // l is the chain length
5.       for i=1 to n-l+1
6.          j=i+l-1
7.          m[i,j]=∞
8.          for k=i to j-1
9.             q=m[i,k]+m[k+1,j]+p_{i-1}p_k p_j
10.            if q < m[i,j]
11.               m[i,j] = q
12.               s[i,j] = k
13.   return m and s
```

i=1 to n-l+1 (n-4+1)

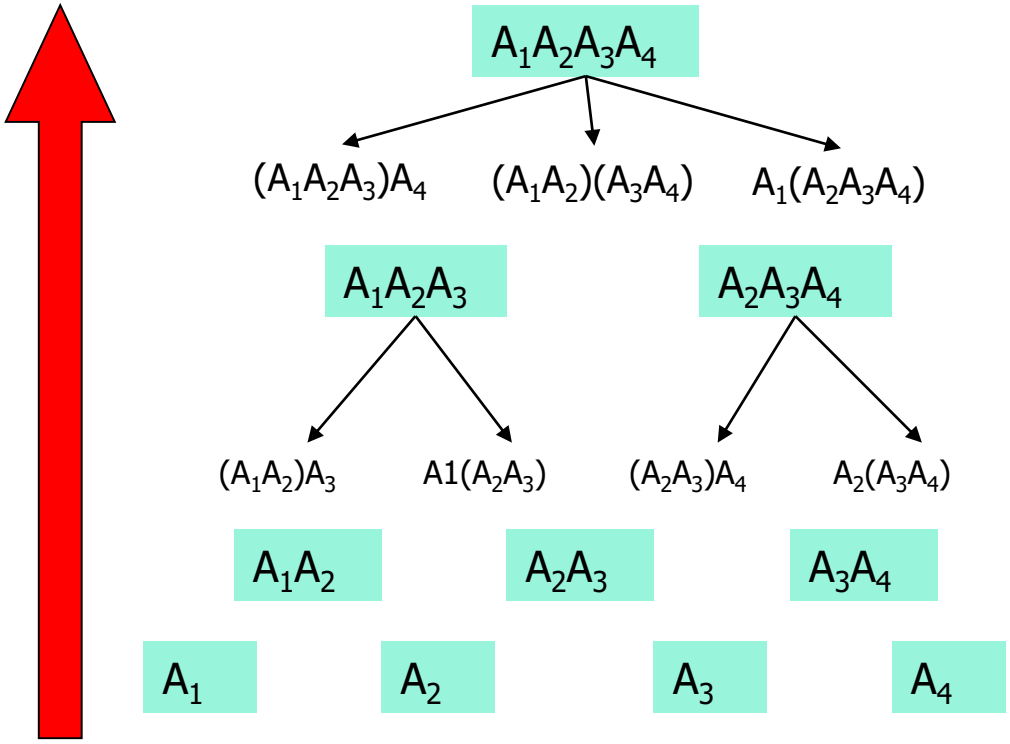
	i=1	i=2	…	i=n-3	i=n-2	i=n-1
l = 2	m[1,2]	m[2,3]	…	m[n-3,n-2]	m[n-2,n-1]	m[n-1,n]
l = 3	m[1,3]	m[2,4]	…	m[n-3,n-1]	m[n-2,n]	X
l = 4	m[1,4]	m[2,5]	…	m[n-3,n]	X	X

Step 3: Computing the Optimal Cost

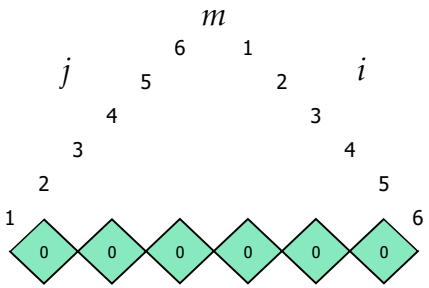
```
MATRIX-CHAIN-ORDER(p) // p = <p0, p1, …, pn>
1.    n= p.length-1
2.    for i=1 to n
3.       m[i,i]=0
4.    for l=2 to n              // l is the chain length
5.       for i=1 to n-l+1
6.          j=i+l-1
7.          m[i,j]=∞
8.          for k=i to j-1
9.             q=m[i,k]+m[k+1,j]+p_{i-1}p_k p_j
10.            if q < m[i,j]
11.               m[i,j] = q
12.               s[i,j] = k
13.   return m and s
```

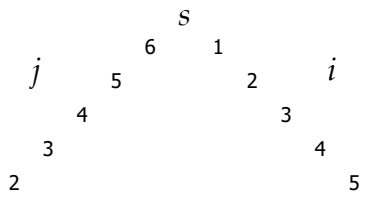
	i=1	i=2	…	i=n-3	i=n-2	i=n-1
l = 2	m[1,2]	m[2,3]	…	m[n-3,n-2]	m[n-2,n-1]	m[n-1,n]
l = 3	m[1,3]	m[2,4]	…	m[n-3,n-1]	m[n-2,n]	X
l = 4	m[1,4]	m[2,5]	…	m[n-3,n]	X	X
…						

Matrix-Chain Multiplication – How Does the Algorithm Work?



Matrix-Chain Multiplication – How Does the Algorithm Work?



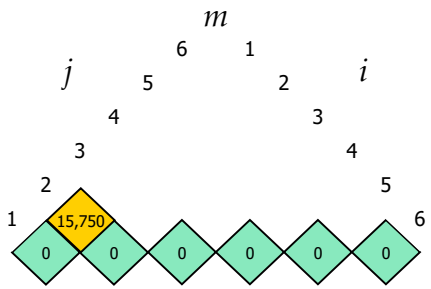


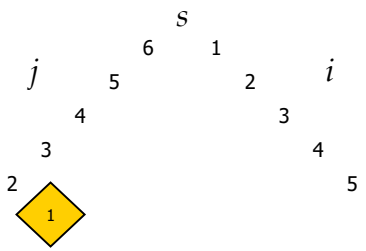
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\}$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





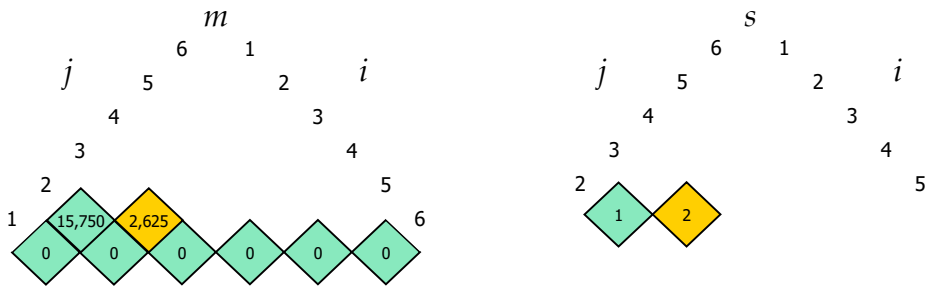
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

$$m[1,2] = m[1,1] + m[2,2] + p_0 p_1 p_2 = 15750$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?



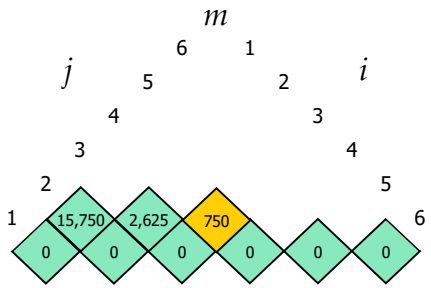
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

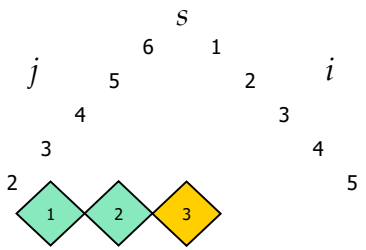
$$m[2,3] = m[2,2] + m[3,3] + p_1 p_2 p_3 = 2625$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





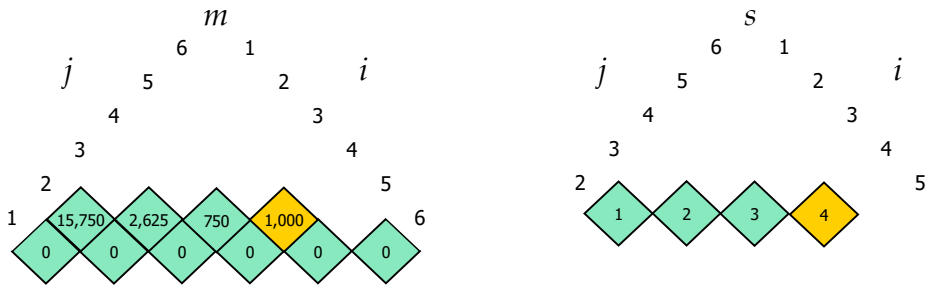
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

$$m[3,4] = m[3,3] + m[4,4] + p_2 p_3 p_4 = 750$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30×35	35×15	15×5	5×10	10×20	20×25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?



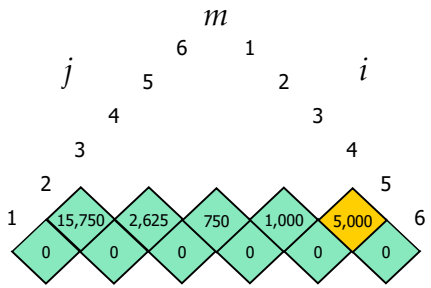
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

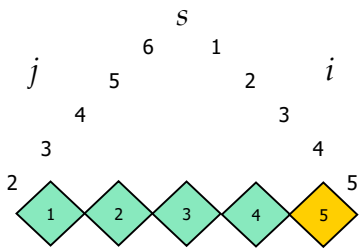
$$m[4,5] = m[4,4] + m[5,5] + p_3 p_4 p_5 = 1000$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





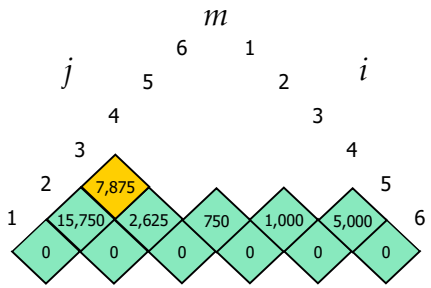
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

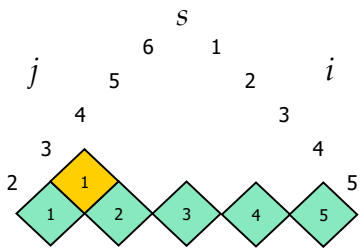
$$m[5,6] = m[5,5] + m[6,6] + p_4 p_5 p_6 = 5000$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30×35	35×15	15×5	5×10	10×20	20×25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





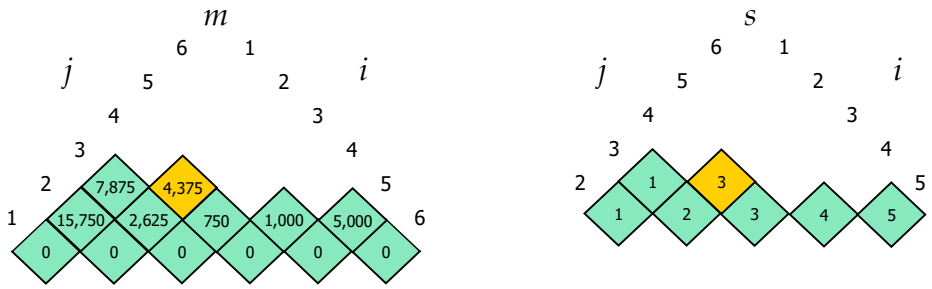
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

$$m[1,3] = min \begin{cases} m[1,1] + m[2,3] + p_0 p_1 p_3 = 0 + 2625 + 5250 = 7875 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 = 15750 + 0 + 2250 = 18000 \end{cases}$$
$$= 7875$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?



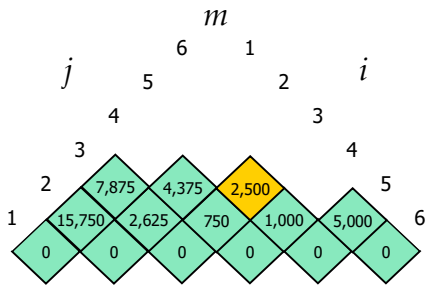
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

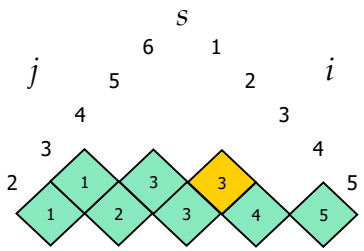
$$m[2,4] = min \begin{cases} m[2,2] + m[3,4] + p_1p_2p_4 = 0 + 750 + 5250 = 6000 \\ m[2,3] + m[4,4] + p_1p_3p_4 = 2625 + 0 + 1750 = 4375 \end{cases}$$
$$= 4375$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





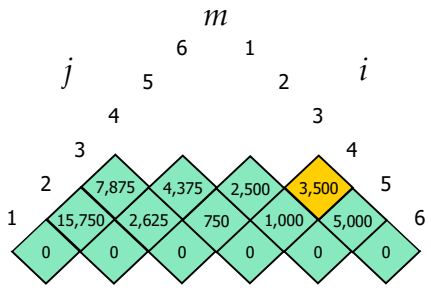
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

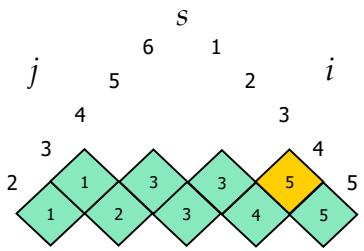
$$m[3,5] = min \begin{cases} m[3,3] + m[4,5] + p_2 p_3 p_5 = 0 + 1000 + 1500 = 2500 \\ m[3,4] + m[5,5] + p_2 p_4 p_5 = 750 + 0 + 3000 = 3750 \end{cases}$$
$$= 2500$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





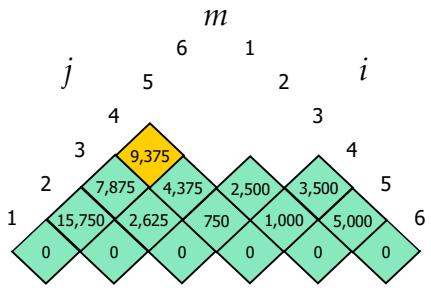
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

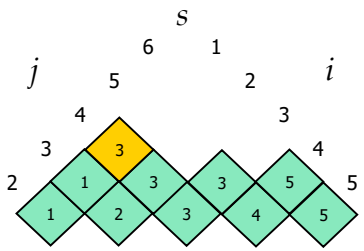
$$m[4,6] = min \begin{cases} m[4,4] + m[5,6] + p_3 p_4 p_6 = 0 + 5000 + 1250 = 6250 \\ m[4,5] + m[6,6] + p_3 p_5 p_6 = 1000 + 0 + 2500 = 3500 \end{cases}$$
$$= 3500$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_5
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





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

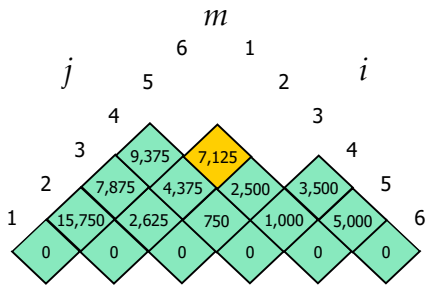
$$m[1,4] = min \begin{cases} m[1,1] + m[2,4] + p_0 p_1 p_4 = 0 + 4375 + 10500 = 14875 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 = 15750 + 750 + 4500 = 21000 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 = 7875 + 0 + 1500 = 9375 \end{cases}$$

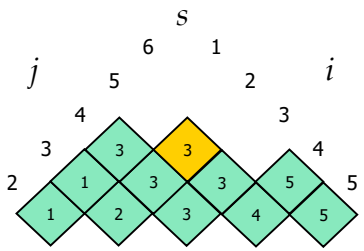
$$= 9375$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





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

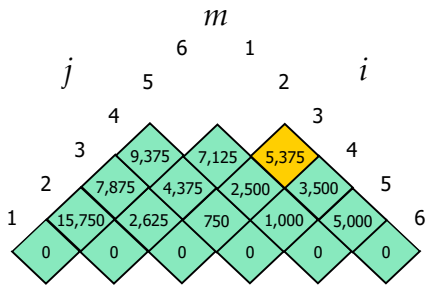
$$m[2,5] = min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 10500 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 3500 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 7000 = 11375 \end{cases}$$

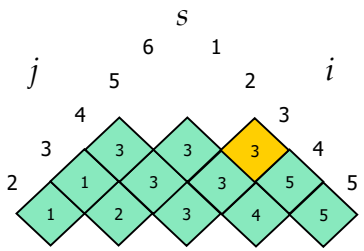
$$= 7125$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





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

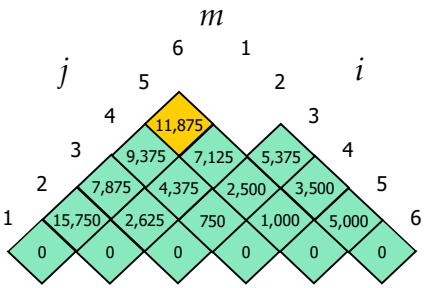
$$m[3,6] = min \begin{cases} m[3,3] + m[4,6] + p_2p_3p_6 = 0 + 3500 + 1875 = 5375 \\ m[3,4] + m[5,6] + p_2p_4p_6 = 750 + 5000 + 3750 = 9500 \\ m[3,5] + m[6,6] + p_2p_5p_6 = 2500 + 0 + 7500 = 10000 \end{cases}$$

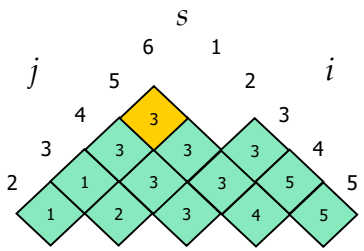
$$= 5375$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





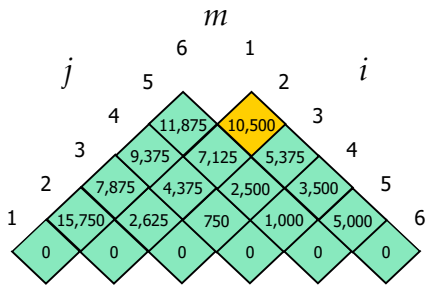
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

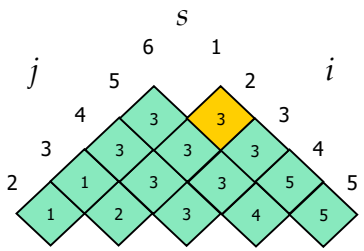
$$m[1,5] = min \begin{cases} m[1,1] + m[2,5] + p_0p_1p_5 = 0 + 7125 + 21000 = 28125 \\ m[1,2] + m[3,5] + p_0p_2p_5 = 15750 + 2500 + 9000 = 27250 \\ m[1,3] + m[4,5] + p_0p_3p_5 = 7875 + 1000 + 3000 = 11875 \\ m[1,4] + m[5,5] + p_0p_4p_5 = 9375 + 0 + 6000 = 15375 \end{cases}$$
$$= 11875$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





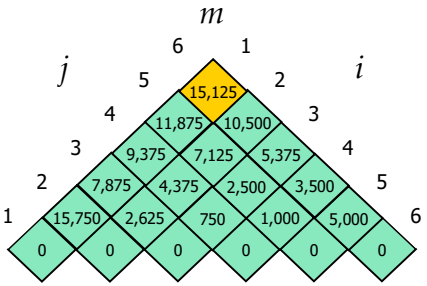
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

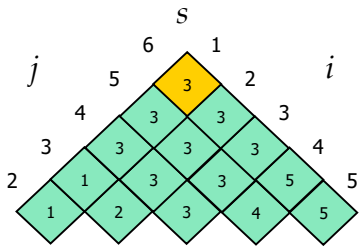
$$m[2,6] = min \begin{cases} m[2,2] + m[3,6] + p_1p_2p_6 = 0 + 5375 + 13125 = 18500 \\ m[2,3] + m[4,6] + p_1p_3p_6 = 2625 + 3500 + 4375 = 10500 \\ m[2,4] + m[5,6] + p_1p_4p_6 = 4375 + 5000 + 8750 = 18125 \\ m[2,5] + m[6,6] + p_1p_5p_6 = 7125 + 0 + 17500 = 24625 \end{cases}$$
$$= 10500$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30×35	35×15	15×5	5×10	10×20	20×25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





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

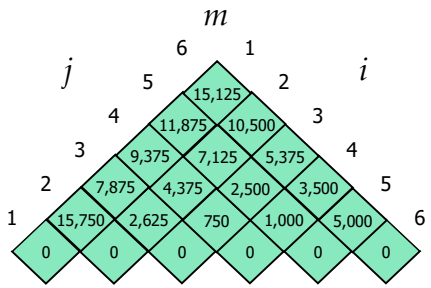
$$m[1,6] = min \begin{cases} m[1,1] + m[2,6] + p_0 p_1 p_6 = 0 + 10500 + 26250 = 36750 \\ m[1,2] + m[3,6] + p_0 p_2 p_6 = 15750 + 5375 + 11250 = 32375 \\ m[1,3] + m[4,6] + p_0 p_3 p_6 = 7875 + 3500 + 3750 = 15125 \\ m[1,4] + m[5,6] + p_0 p_4 p_6 = 9375 + 5000 + 7500 = 21875 \\ m[1,5] + m[6,6] + p_0 p_5 p_6 = 11875 + 0 + 15000 = 26875 \end{cases}$$

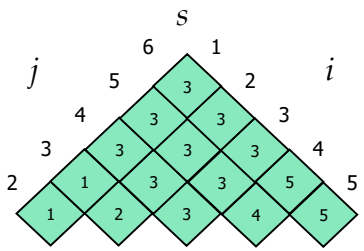
$$= 15125$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – How Does the Algorithm Work?





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

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Step 4: Constructing an Optimal Solution

- Although RECURSIVE-MATRIX-CHAIN provide the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices.
- Each entry s[i, j] records a value of k such that an optimal parenthesization of $A_iA_{i+1}...A_j$ splits the product between A_k and A_{k+1}.
- Thus, we know that the final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..s[1,n]}A_{s[1,n]+1..n}$.
- We can determine the earlier matrix multiplications recursively.
  - s[1, s[1,n]] determines the last matrix multiplication when computing $A_{1..s[1,n]}$
  - s[s[1,n]+1,n] determines the last matrix multiplication when computing $A_{s[1,n]+1..n}$.

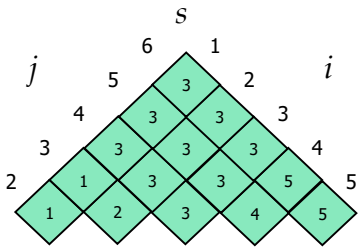
Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
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 ")"
```

Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
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 ")"
```

- PRINT-OPTIMAL-PARENS(s,1,6)

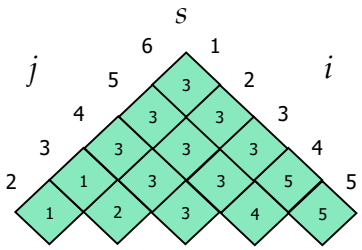


Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
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 ")"
```

i:1
j:6
Output:

(i,j) = (1,6)



- PRINT-OPTIMAL-PARENS(s,1,6)

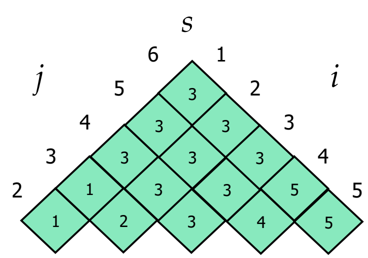
Constructing an Optimal Solution

```
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 ")"
```

i:1
j:6
Output: (

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



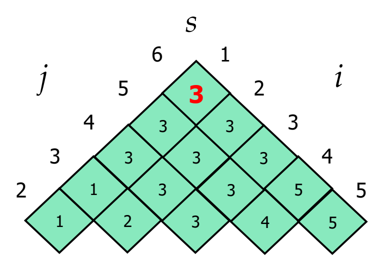
Constructing an Optimal Solution

```
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,1,3)
        PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
        print ")"
```

i:1
j:6
Output: (

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

PRINT-OPTIMAL-PARENS(s,i,j)

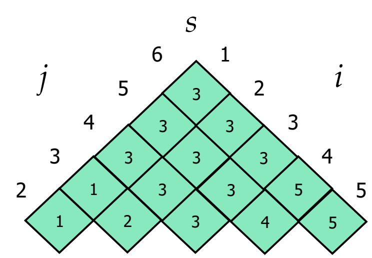
1. **if** i == j
2. print "A"$_i$
3. **else**
4. print "("
5. PRINT-OPTIMAL-PARENS(s,i,s[i,j])
6. PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
7. print ")"

i:1
j:3
Output: (

(i,j) = (1,3)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

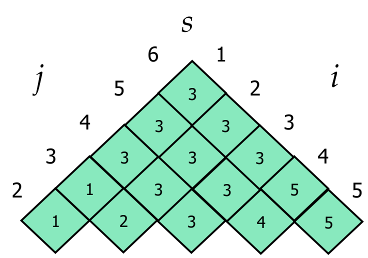
```
PRINT-OPTIMAL-PARENS(s,i,j)
    if i == j
        print "A"_i
    else
        print "("
        PRINT-OPTIMAL-PARENS(s,i,s[i,j])
        PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
        print ")"
```

i:1
j:3
Output: ((

(i,j) = (1,3)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

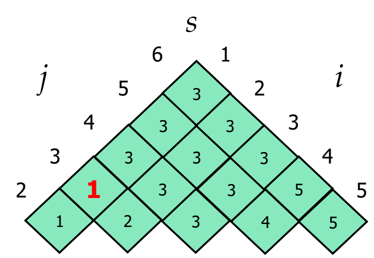
```
PRINT-OPTIMAL-PARENS(s,i,j)
    if i == j
        print "A"_i
    else
        print "("
        PRINT-OPTIMAL-PARENS(s,i,s[i,j])  →  PRINT-OPTIMAL-PARENS(s,1,1)
        PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
        print ")"
```

i:1
j:3
Output: ((

(i,j) = (1,3)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

PRINT-OPTIMAL-PARENS(s,i,j)

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 ")"

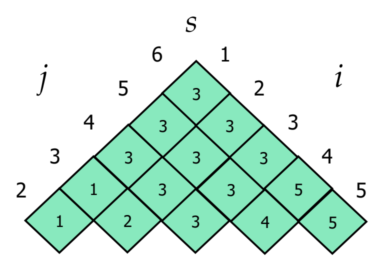
i:1
j:1
Output: ((A_1

(i,j) = (1,1)

(i,j) = (1,3)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

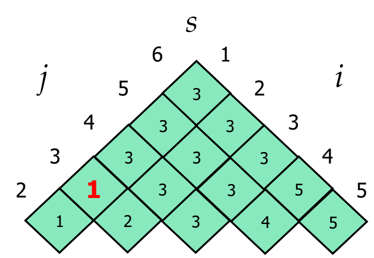
```
PRINT-OPTIMAL-PARENS(s,i,j)
    if  i == j
        print "A"_i
    else
        print "("
        PRINT-OPTIMAL-PARENS(s,i,s[i,j]) return
        PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j) → PRINT-OPTIMAL-PARENS(s,2,3)
        print ")"
```

i:1
j:3
Output: ((A_1

(i,j) = (1,3)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
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 ")"
```

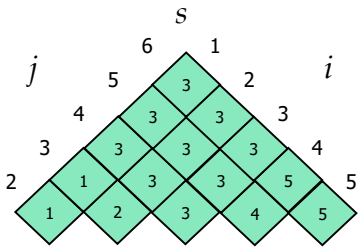
i:2
j:3
Output: ((A$_1$

(i,j) = (2,3)

(i,j) = (1,3)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
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 ")"
```

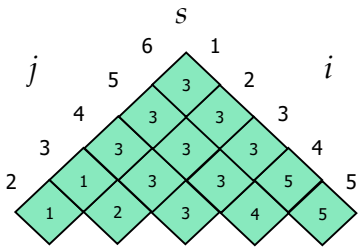
i:2
j:3
Output: ((A$_1$(

(i,j) = (2,3)

(i,j) = (1,3)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

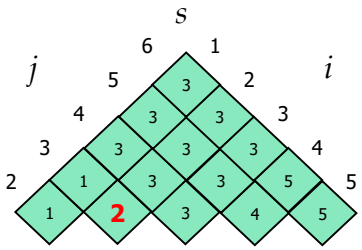
```
PRINT-OPTIMAL-PARENS(s,i,j)
1.   if  i == j
2.      print "A"_i
3.   else
4.      print "("
5.      PRINT-OPTIMAL-PARENS(s,i,s[i,j])
6.      PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
7.      print ")"
```

i:2
j:3
Output: ((A$_1$(

(i,j) = (2,3)
(i,j) = (1,3)
(i,j) = (1,6)

PRINT-OPTIMAL-PARENS(s,2,2)

- PRINT-OPTIMAL-PARENS(s,1,6)



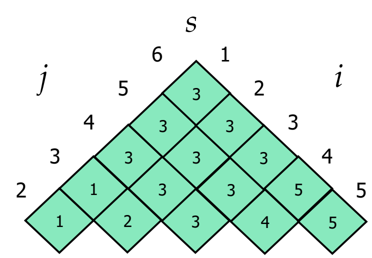
Constructing an Optimal Solution

PRINT-OPTIMAL-PARENS(s,i,j)
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 ")"

i:2
j:2
Output: ((A_1(A_2

(i,j) = (2,2)
(i,j) = (2,3)
(i,j) = (1,3)
(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
1.   if  i == j
2.      print "A"_i
3.   else
4.      print "("
5.      PRINT-OPTIMAL-PARENS(s,i,s[i,j]) return
6.      PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
7.      print ")"
```

i:2
j:3
Output: ((A$_1$(A$_2$

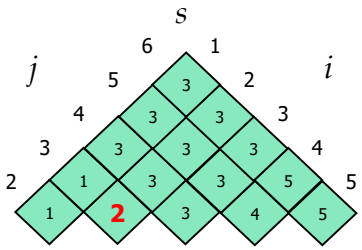
(i,j) = (2,3)

(i,j) = (1,3)

(i,j) = (1,6)

→ PRINT-OPTIMAL-PARENS(s,3,3)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

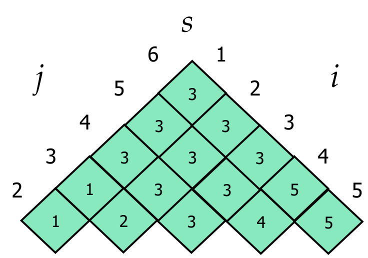
PRINT-OPTIMAL-PARENS(s,i,j)

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 ")"

i:3
j:3
Output: ((A_1($A_2$$A_3$

(i,j) = (3,3)
(i,j) = (2,3)
(i,j) = (1,3)
(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

```
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)return
    print ")"
```

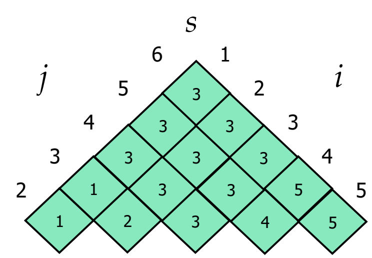
i:2
j:3
Output: ((A_1($A_2$$A_3$)

(i,j) = (2,3)

(i,j) = (1,3)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

```
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)return
        print ")"
```

i:1

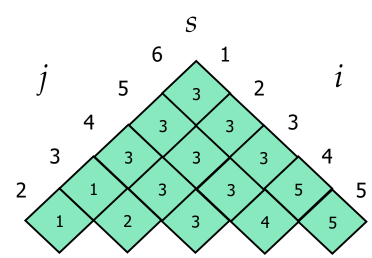
j:3

Output: $((A_1(A_2A_3))$

(i,j) = (1,3)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

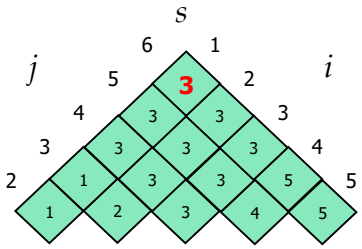
```
PRINT-OPTIMAL-PARENS(s,i,j)
1.   if  i == j
2.      print "A"_i
3.   else
4.      print "("
5.      PRINT-OPTIMAL-PARENS(s,i,s[i,j]) return
6.      PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
7.      print ")"
```

i:1
j:6
Output: $((A_1(A_2A_3))$

(i,j) = (1,6)

→ PRINT-OPTIMAL-PARENS(s,4,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

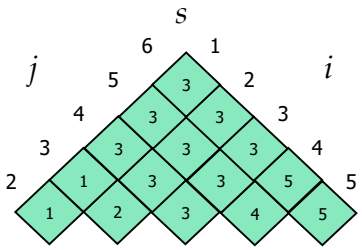
```
PRINT-OPTIMAL-PARENS(s,i,j)
1.   if  i == j
2.      print "A"_i
3.   else
4.      print "("
5.      PRINT-OPTIMAL-PARENS(s,i,s[i,j])
6.      PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
7.      print ")"
```

i:4
j:6
Output: $((A_1(A_2A_3))$

(i,j) = (4,6)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

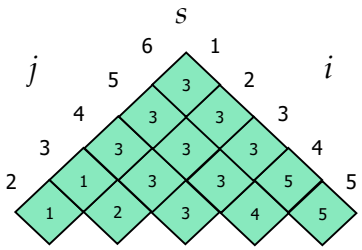
```
PRINT-OPTIMAL-PARENS(s,i,j)
1.   if  i == j
2.      print "A"_i
3.   else
4.      print "("
5.      PRINT-OPTIMAL-PARENS(s,i,s[i,j])
6.      PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
7.      print ")"
```

i:4
j:6
Output: $((A_1(A_2A_3))($

(i,j) = (4,6)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
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 ")"
```

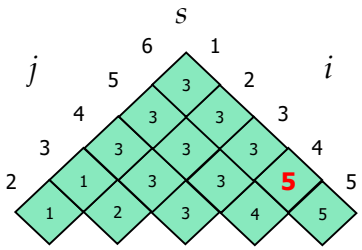
i:4
j:6
Output: $((A_1(A_2A_3))($

(i,j) = (4,6)

(i,j) = (1,6)

PRINT-OPTIMAL-PARENS(s,4,5)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
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 ")"
```

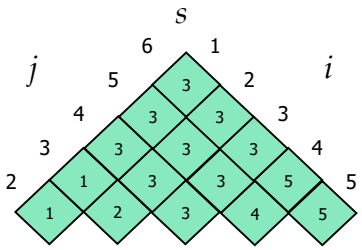
i:4
j:5
Output: $((A_1(A_2A_3))($

(i,j) = (4,5)

(i,j) = (4,6)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
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 ")"
```

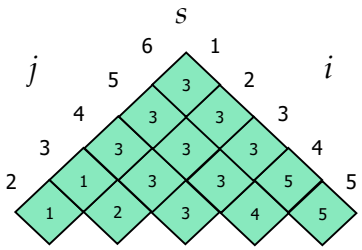
i:4
j:5
Output: $((A_1(A_2A_3))($

(i,j) = (4,5)

(i,j) = (4,6)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



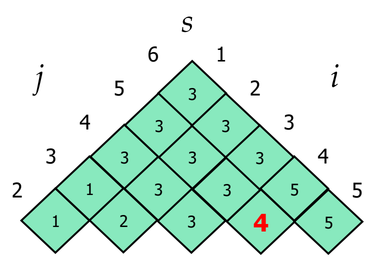
Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
1.  if  i == j
2.      print "A"_i
3.  else
4.      print "("
5.      PRINT-OPTIMAL-PARENS(s,i,s[i,j])  →  PRINT-OPTIMAL-PARENS(s,4,4)
6.      PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
7.      print ")"
```

i:4
j:5
Output: ((A_1(A_2A_3))((

(i,j) = (4,5)
(i,j) = (4,6)
(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



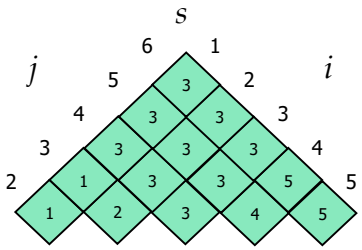
Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
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 ")"
```

i:4
j:4
Output: $((A_1(A_2A_3))((A_4$

(i,j) = (4,4)
(i,j) = (4,5)
(i,j) = (4,6)
(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



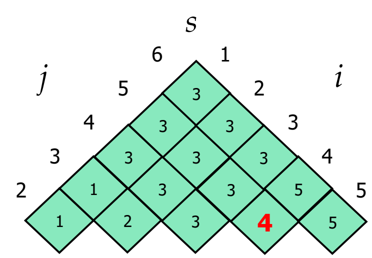
Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
if  i == j
    print "A"i
else
    print "("
    PRINT-OPTIMAL-PARENS(s,i,s[i,j]) return
    PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j) → PRINT-OPTIMAL-PARENS(s,5,5)
    print ")"
```

i:4
j:5
Output: $((A_1(A_2A_3))((A_4$

(i,j) = (4,5)
(i,j) = (4,6)
(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



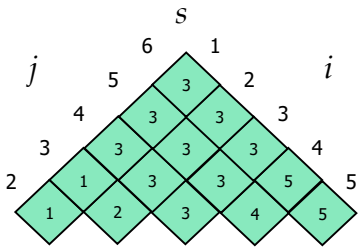
Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
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 ")"
```

i:5
j:5
Output: $((A_1(A_2A_3))((A_4A_5$

(i,j) = (5,5)
(i,j) = (4,5)
(i,j) = (4,6)
(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

```
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) return
        print ")"
```

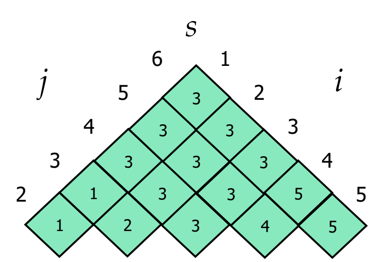
i:4
j:5
Output: $((A_1(A_2A_3))((A_4A_5)$

(i,j) = (4,5)

(i,j) = (4,6)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
1.   if  i == j
2.      print "A"_i
3.   else
4.      print "("
5.      PRINT-OPTIMAL-PARENS(s,i,s[i,j]) return
6.      PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
7.      print ")"
```

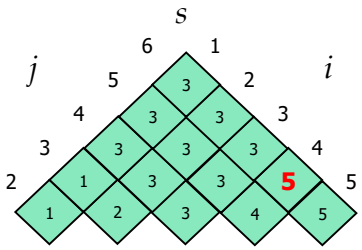
i:4
j:6
Output: $((A_1(A_2A_3))((A_4A_5)$

(i,j) = (4,6)

(i,j) = (1,6)

→ PRINT-OPTIMAL-PARENS(s,6,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



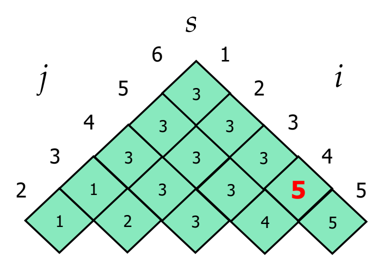
Constructing an Optimal Solution

```
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 ")"
```

i:4
j:6
Output: $((A_1(A_2A_3))((A_4A_5)A_6$

(i,j) = (6,6)
(i,j) = (4,6)
(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



Constructing an Optimal Solution

PRINT-OPTIMAL-PARENS(s,i,j)

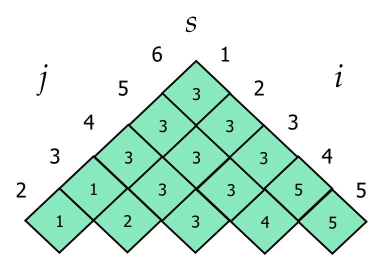
1. **if** i == j
2. print "A"$_i$
3. **else**
4. print "("
5. PRINT-OPTIMAL-PARENS(s,i,s[i,j])
6. PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)return
7. print ")"

i:4
j:6
Output: $((A_1(A_2A_3))((A_4A_5)A_6)$

(i,j) = (4,6)

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)



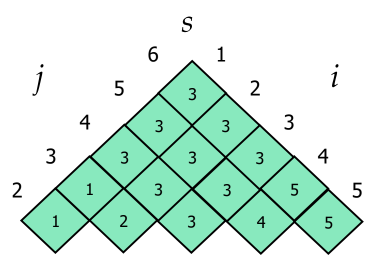
Constructing an Optimal Solution

```
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)return
        print ")"
```

i:1
j:6
Output: $((A_1(A_2A_3))((A_4A_5)A_6))$

(i,j) = (1,6)

- PRINT-OPTIMAL-PARENS(s,1,6)

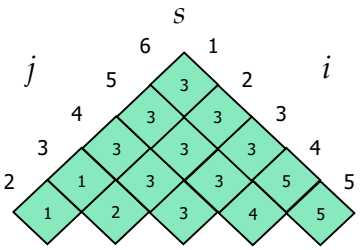


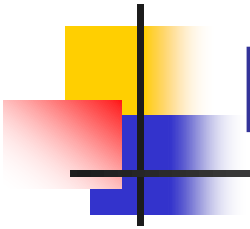
Constructing an Optimal Solution

```
PRINT-OPTIMAL-PARENS(s,i,j)
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 ")"
```

Output: $((A_1(A_2A_3))((A_4A_5)A_6))$

- PRINT-OPTIMAL-PARENS(s,1,6)
  - $((A_1(A_2A_3))((A_4A_5)A_6))$ is an optimal solution

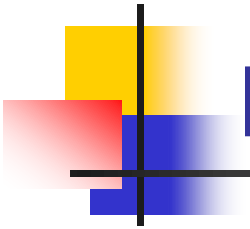




Dynamic Programming

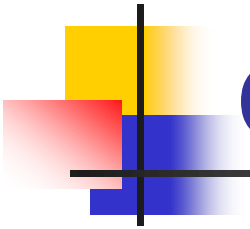
Dynamic programming is applicable when the subproblems are independent

- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution in a bottom-up fashion
- Construct an optimal solution from computed information



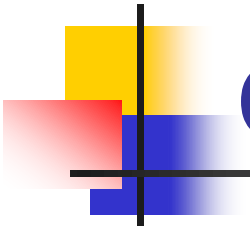
Elements of Dynamic Programming

- When should we look for a dynamic programming solution to a problem?
- In order for dynamic programming to apply, the problem must have
  - Optimal substructure
  - Overlapping subproblems



Optimal Substructure

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.
- Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply.
- In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems.
- Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.



Optimal Substructure

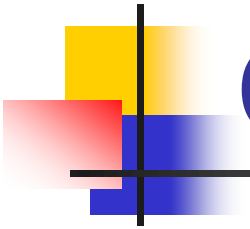
- We discovered the optimal substructure in both of the problems we have examined so far.
  - In rod cutting, we observed that the optimal way of cutting up a rod of length n (if we make any cuts at all) involves optimally cutting up the two pieces resulting from the first cut.
  - In matrix multiplication, we observed that an optimal parenthesization of $A_iA_{i+1}...A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problems of parenthesizing $A_iA_{i+1}...A_k$ and $A_{k+1}...A_j$.

Optimal Substructure

- Show that a solution to the problem consists of making a choice, such as choosing an initial cut in a rod or choosing an index at which to split the matrix chain.
- Assume that you are given the choice that leads to an optimal solution.
- Given this choice, determine which subproblems ensue and how to best characterize the resulting space of subproblems.
- Show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique.
- By supposing that each of the subproblem solutions is not optimal and then deriving a contradiction.
  - e.g.,) By "cutting out" the nonoptimal solution to each subproblem and "pasting in" the optimal one, Show that you get a better solution to the original problem, thus contradicting your supposition.

Running Time

- The running time of a dynamic programming algorithm depends on the product of two factors
  - the number of subproblems overall
  - how many choices we look at for each subproblem.
- In rod cutting, we had $\Theta(n)$ subproblems overall, and at most n choices to examine for each, yielding an $\Theta(n^2)$ running time.
- Matrix-chain multiplication had $\Theta(n^2)$ subproblems overall, and in each we had at most n-1 choices, giving an $O(n^3)$ running time.



Overlapping Subproblems

- The space of subproblems must be "small" in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems.
- When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has overlapping subproblems.
- Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

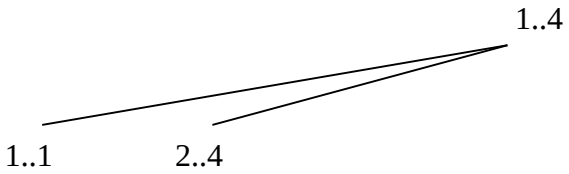
Overlapping Subproblems

The recursion tree for RECURSIVE-MATRIX-CHAIN(p,1,4).

1..4

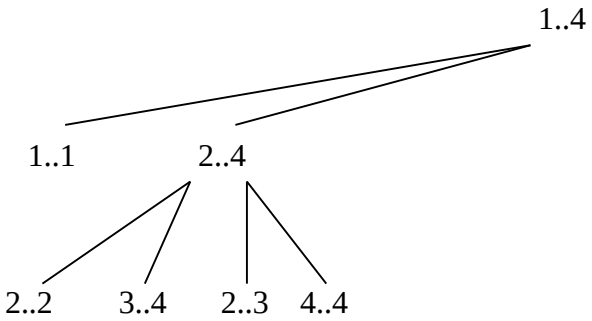
Overlapping Subproblems

The recursion tree for RECURSIVE-MATRIX-CHAIN(p,1,4).



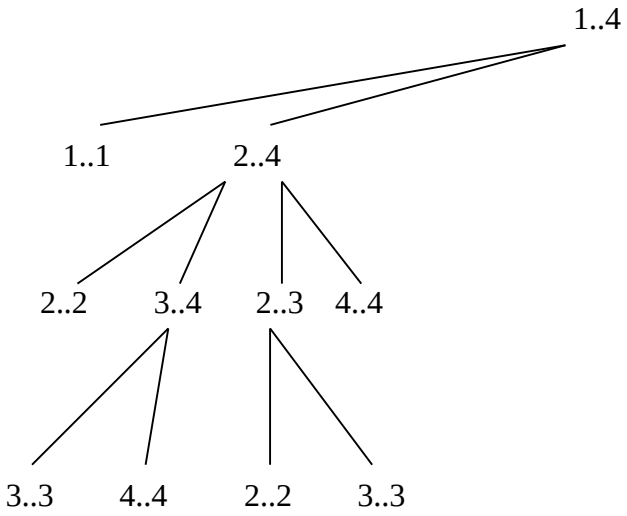
Overlapping Subproblems

The recursion tree for RECURSIVE-MATRIX-CHAIN(p,1,4).



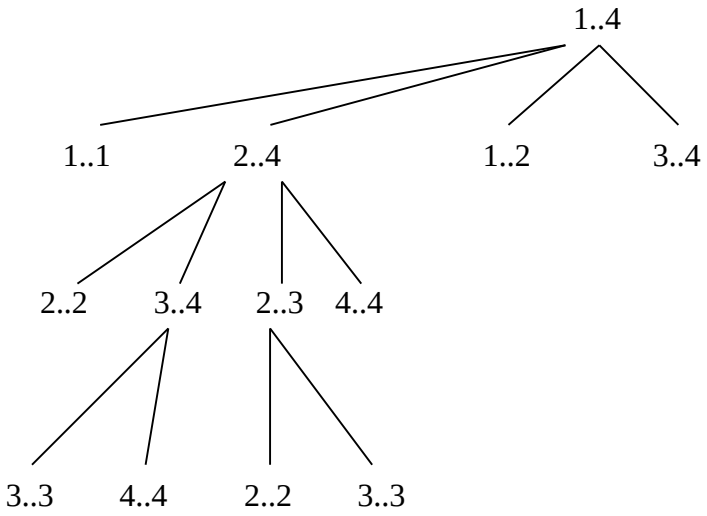
Overlapping Subproblems

The recursion tree for RECURSIVE-MATRIX-CHAIN(p,1,4).



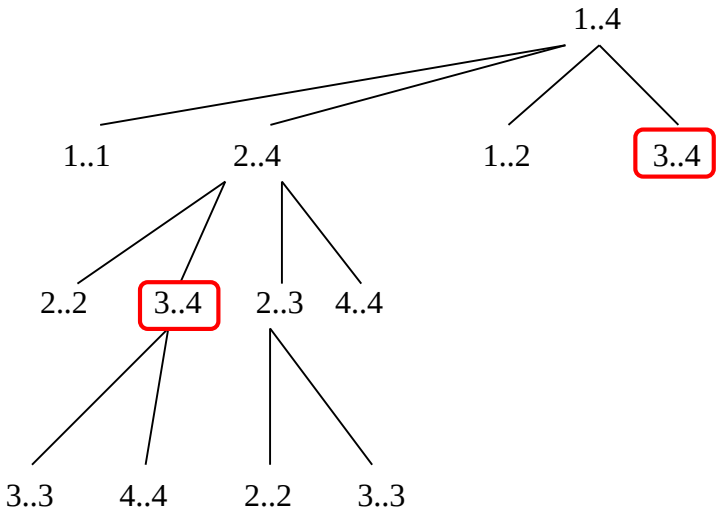
Overlapping Subproblems

The recursion tree for RECURSIVE-MATRIX-CHAIN(p,1,4).



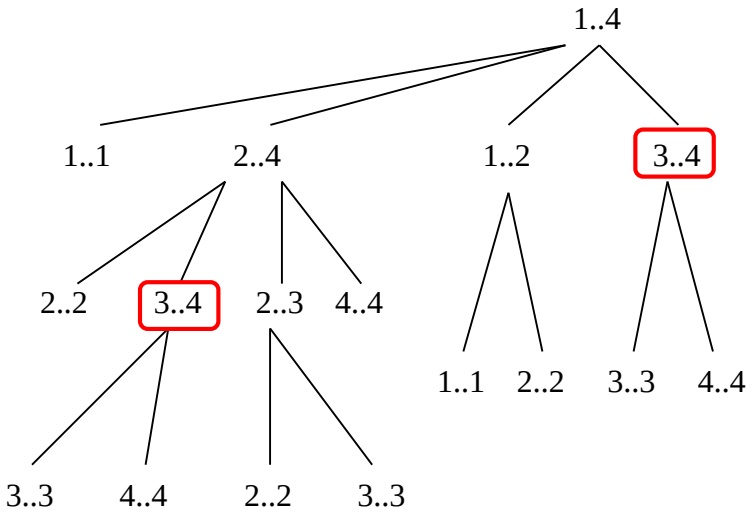
Overlapping Subproblems

The recursion tree for RECURSIVE-MATRIX-CHAIN(p,1,4).



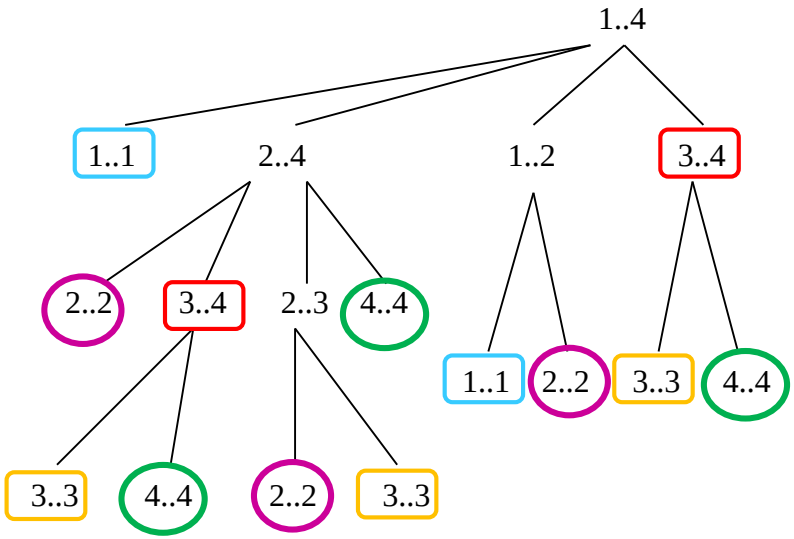
Overlapping Subproblems

The recursion tree for RECURSIVE-MATRIX-CHAIN(p,1,4).



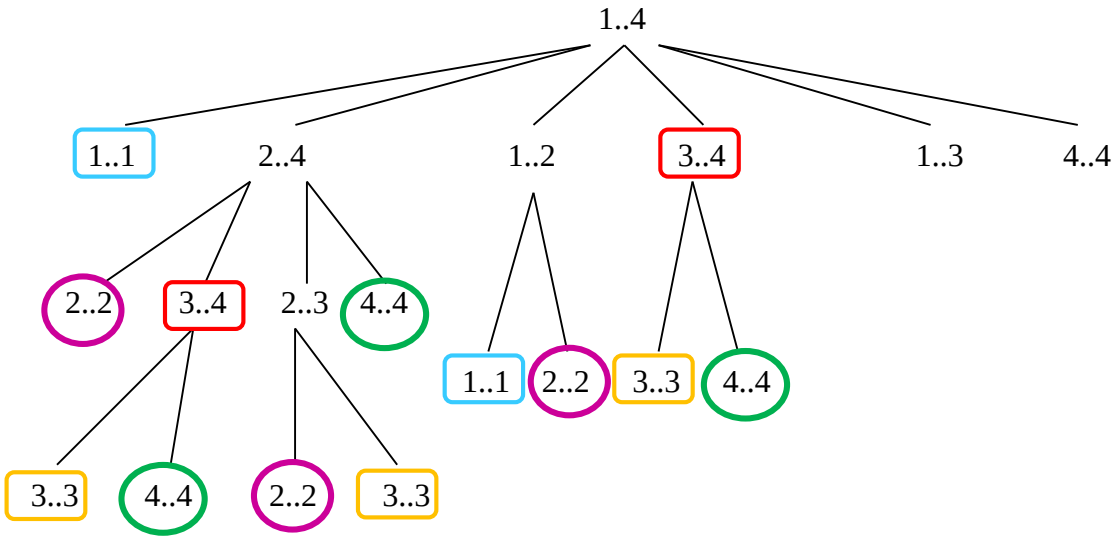
Overlapping Subproblems

The recursion tree for RECURSIVE-MATRIX-CHAIN(p,1,4).



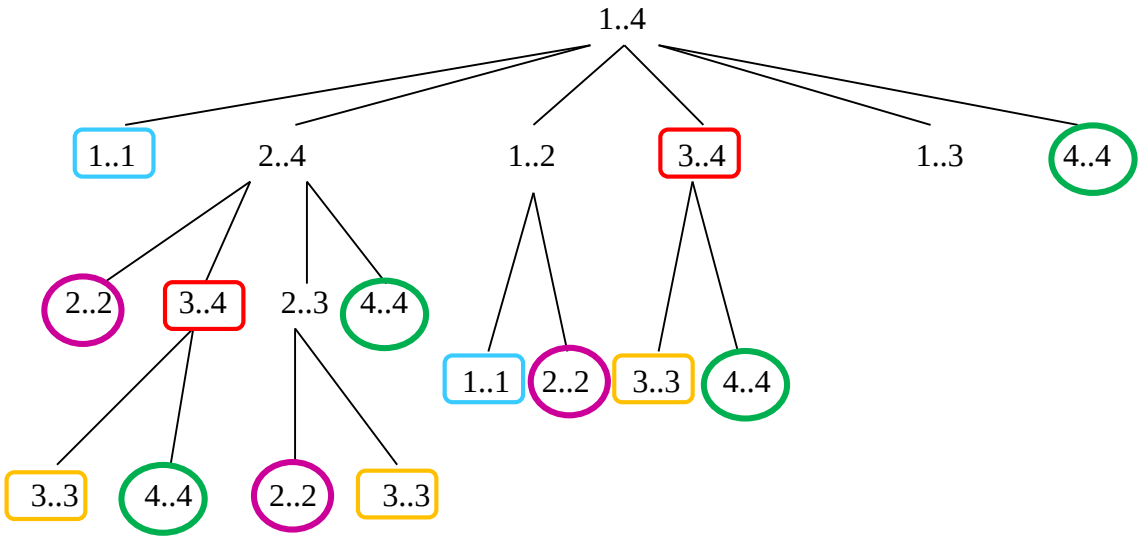
Overlapping Subproblems

The recursion tree for RECURSIVE-MATRIX-CHAIN(p,1,4).



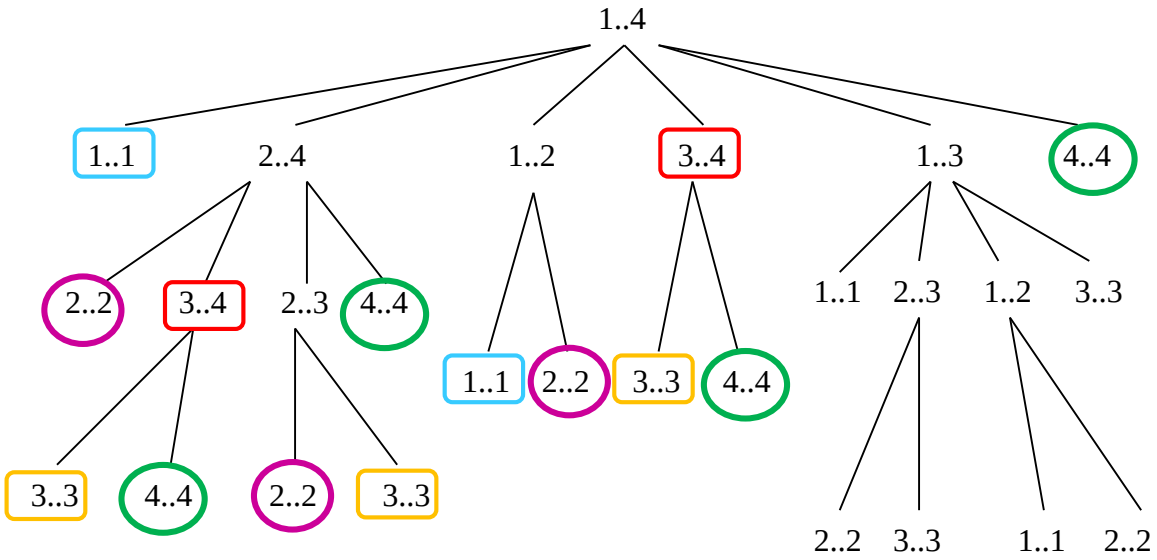
Overlapping Subproblems

The recursion tree for RECURSIVE-MATRIX-CHAIN(p,1,4).



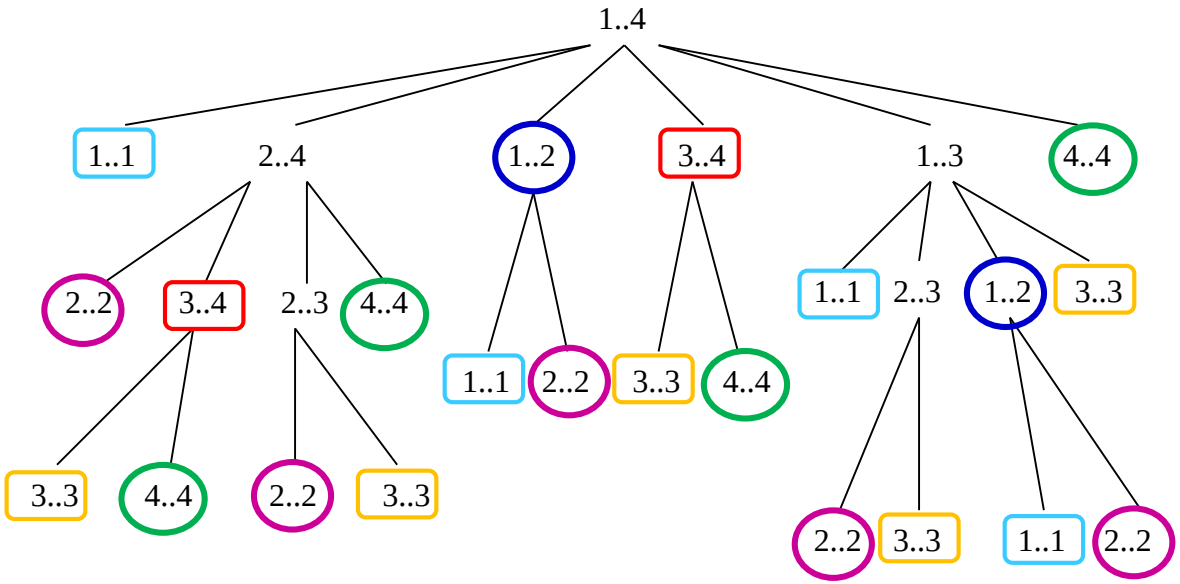
Overlapping Subproblems

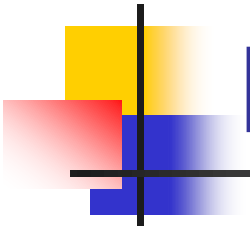
The recursion tree for RECURSIVE-MATRIX-CHAIN(p,1,4).



Overlapping Subproblems

The recursion tree for RECURSIVE-MATRIX-CHAIN(p,1,4).





Recursive Solution

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1.   if i == j
2.       return 0
3.   m[i,j] = ∞
4.   for k = i to j-1
5.       q = RECURSIVE-MATRIX-CHAIN (p, i, k)
6.             +RECURSIVE-MATRIX-CHAIN(p, k+1, j)
7.             +p_{i-1} p_k p_j
8.       if q < m[i,j]
9.          m[i,j] = q
10.  return m [i, j]
```

Recurrence for Time Complexity?

- $T(1) \geq 1$
- $T(n) \geq 1 + \sum_{k=0}^{n-1}(T(k) + T(n-k) + 1)\ for\ n > 1$

  $\geq 2\sum_{k=0}^{n-1} T(k) + n$
- We will show that $T(n) \geq 2^{n-1}$ for $n \geq 1$.
  - Basis: $T(1) \geq 1 = 2^0$
  - Inductively, for $n \geq 2$, we have
  - $T(n) \geq 2\sum_{i=1}^{n-1} 2^{i-1} + n$

    $= 2\sum_{i=0}^{n-2} 2^i + n$

    $= 2(2^{n-1} - 1) + n = 2^n - 2 + n$

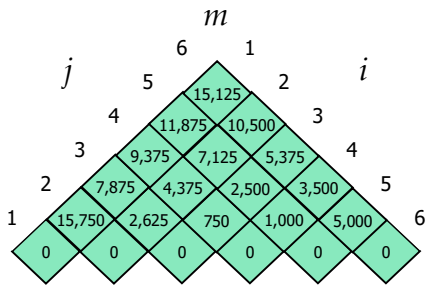
    $= 2^{n-1} + 2^{n-1} - 2 + n$

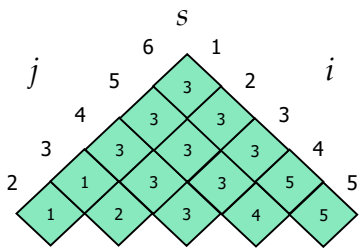
    $\geq 2^{n-1}$

Reconstructing an Optimal Solution

- As a practical matter, we often store which choice we made in each subproblem in a table so that we do not have to reconstruct this information from the costs that we stored.
- For matrix-chain multiplication, suppose that we did not maintain the s[i, j] table and have filled in only the table m[i,j] containing optimal subproblem costs.
- We have to choose from among (j – i) possibilities when we determine which subproblems to use in an optimal solution to parenthesizing $A_iA_{i+1}...A_j$, and j - i is not a constant.
- Therefore, it would take $\Theta(n)$ time to reconstruct which subproblems we chose for a solution to a given problem.
- By storing in s[i, j] the index of the matrix at which we split product $A_iA_{i+1}...A_j$, we can reconstruct each choice in O(1) time.

Matrix-Chain Multiplication – Reconstructing an Optimal Solution



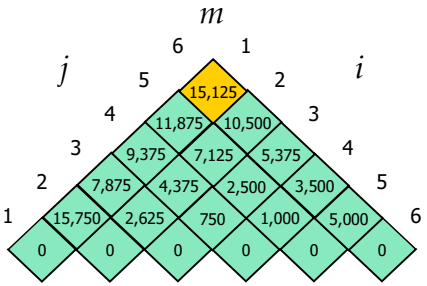


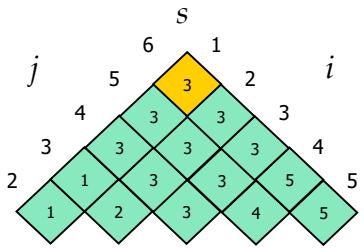
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30×35	35×15	15×5	5×10	10×20	20×25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25

Matrix-Chain Multiplication – Reconstructing an Optimal Solution





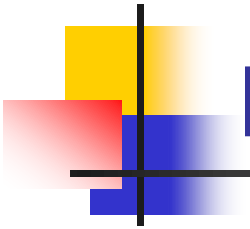
$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$$

$$m[1,6] = min \begin{cases} m[1,1] + m[2,6] + p_0p_1p_6 = 0 + 10500 + 26250 = 36750 \\ m[1,2] + m[3,6] + p_0p_2p_6 = 15750 + 5375 + 11250 = 32375 \\ m[1,3] + m[4,6] + p_0p_3p_6 = 7875 + 3500 + 3750 = 15125 \\ m[1,4] + m[5,6] + p_0p_4p_6 = 9375 + 5000 + 7500 = 21875 \\ m[1,5] + m[6,6] + p_0p_5p_6 = 11875 + 0 + 15000 = 26875 \end{cases}$$

$$= 15125$$

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

p_0	p_1	p_2	p_3	p_4	p_5	p_6
30	35	15	5	10	20	25



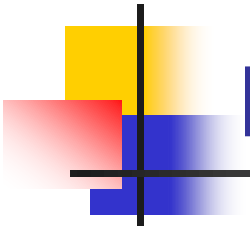
Memoization

- A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem.
- Each table entry initially contains a special value to indicate that the entry has yet to be filled in.
- When the subproblem is first encountered as the recursive algorithm unfolds, its solution is computed and then stored in the table.
- Each subsequent time that we encounter this subproblem, we simply look up the value stored in the table and return it.

Recursive Solution with Memoization

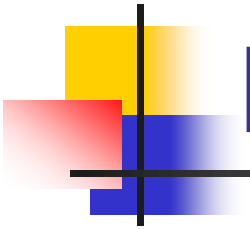
```
MEMOIZED-MATRIX-CHAIN(p)
1.  n=p.legnth-1
2.  let m[1..n, 1..n] be a new table
3.  for i = 1 to n
4.      for j = i to n
5.          m[i,j] = ∞
6.  return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
1.  if m[i, j] < ∞
2.      return m[i, j]
3.  if i==j
4.      m[i,j]=0
5.  else
6.      for k = i to j-1
7.          q = LOOKUP-CHAIN (m, p, i, k)
8.                +LOOKUP-CHAIN (m, p, k+1, j)
9.                +p_{i-1}p_k p_j
10.         if q < m[i,j]
11.             m[i,j] = q
12. return m [i, j]
```



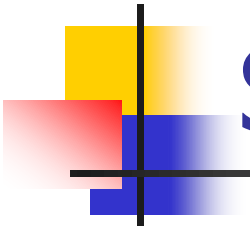
Matrix-chain Multiplication Problem

- We can solve the matrix-chain multiplication problem by either a top-down, memoized dynamic programming algorithm or a bottom-up dynamic programming algorithm in $O(n^3)$ time.
- Both methods take advantage of the overlapping-subproblems property.
- There are only $\Theta(n^2)$ distinct subproblems in total, and either of these methods computes the solution to each subproblem only once.
- Without memoization, the natural recursive algorithm runs in exponential time, since solved subproblem are repeatedly solved.



Dynamic Programming

- If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms the corresponding top-down memoized algorithm by a constant factor, because
  - The bottom-up algorithm has no overhead for recursion and less overhead for maintaining the table.
  - For some problems we can exploit the regular pattern of table accesses in the dynamic programming algorithm to reduce time or space requirements even further.
- Alternatively, if some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required.



Subtleties

- Let us think about these two problems
  - Un-weighted shortest path - find a path from u to v consisting of the fewest edges.
  - Un-weighted longest path - find a simple path from u to v consisting of the most edges.