Algorithms and Data Structures

Introduction to Dynamic Programming and Chain Matrix Multiplication

Dynamic Programming

An technique for solving optimisation problems.

Term attributed to Bellman (1950s).

“Programming” as in “Planning” or “Optimising”.

Dynamic Programming

Dynamic Programming

The paradigm of dynamic programming:

Given a problem P, define a sequence of subproblems, with the following properties:

Dynamic Programming

The paradigm of dynamic programming:

Given a problem P, define a sequence of subproblems, with the following properties:

The subproblems are ordered from the smallest to the largest.

Dynamic Programming

The paradigm of dynamic programming:

Given a problem P, define a sequence of subproblems, with the following properties:

The subproblems are ordered from the smallest to the largest.

The largest problem is our original problem P.

Dynamic Programming

The paradigm of dynamic programming:

Given a problem P, define a sequence of subproblems, with the following properties:

The subproblems are ordered from the smallest to the largest.

The largest problem is our original problem P.

The optimal solution of a subproblem can be constructed from the optimal solutions of sub-sub-problems. (*Optimal Substructure*).

Dynamic Programming

The paradigm of dynamic programming:

Given a problem P, define a sequence of subproblems, with the following properties:

The subproblems are ordered from the smallest to the largest.

The largest problem is our original problem P.

The optimal solution of a subproblem can be constructed from the optimal solutions of sub-sub-problems. (*Optimal Substructure*).

Solve the subproblems from the smallest to the largest. When you solve a subproblem, store the solution (e.g., in an array) and use it to solve the larger subproblems.

Matrix Chain Multiplication

Matrix Chain Multiplication

We have a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices (not necessarily square) to be multiplied.

Matrix Chain Multiplication

We have a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices (not necessarily square) to be multiplied.

The goal is to compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

Matrix Chain Multiplication

We have a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices (not necessarily square) to be multiplied.

The goal is to compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

For that we will use the *standard* algorithm for matrix multiplication:

RECTANGULAR-MATRIX-MULTIPLY (A, B, C, p, q, r)

for $i = 1$ **to** p
 for $j = 1$ **to** r
  for $k = 1$ **to** q
   $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$

Matrix Chain Multiplication

We have a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices (not necessarily square) to be multiplied.

The goal is to compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

For that we will use the *standard* algorithm for matrix multiplication:

```
RECTANGULAR-MATRIX-MULTIPLY(A, B, C, p, q, r)
for i = 1 to p
    for j = 1 to r
        for k = 1 to q
            c_ij = c_ij + a_ik · b_kj
```

The running time is the number of scalar multiplications, i.e., pqr.

Matrix Chain Multiplication

We have a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices (not necessarily square) to be multiplied.

The goal is to compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

Matrix Chain Multiplication

We have a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices (not necessarily square) to be multiplied.

The goal is to compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

The order of multiplication matters!

Matrix Chain Multiplication

We have a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices (not necessarily square) to be multiplied.

The goal is to compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

The order of multiplication matters!

Consider $\langle A_1, A_2, A_3 \rangle$ with dimensions 10×100, 100×5, and 5×50.

Matrix Chain Multiplication

We have a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices (not necessarily square) to be multiplied.

The goal is to compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

The order of multiplication matters!

Consider $\langle A_1, A_2, A_3 \rangle$ with dimensions 10×100, 100×5, and 5×50.

If we do $((A_1 \cdot A_2) \cdot A_3)$ we have $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications + $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications for a total of 7500.

Matrix Chain Multiplication

We have a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices (not necessarily square) to be multiplied.

The goal is to compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

The order of multiplication matters!

Consider $\langle A_1, A_2, A_3 \rangle$ with dimensions 10×100, 100×5, and 5×50.

If we do $((A_1 \cdot A_2) \cdot A_3)$ we have $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications + $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications for a total of 7500.

If instead we do $(A_1 \cdot (A_2 \cdot A_3))$ we have $100 \cdot 5 \cdot 50 = 25000$ scalar multiplications + $10 \cdot 100 \cdot 50 = 50000$ scalar multiplications for a total of 75000.

Full Parenthesisation

A product of matrices is *fully parenthesised* if it is either a single matrix, or a product of two fully parenthesised matrix products, surrounded by parenthesis.

Full Parenthesisation

A product of matrices is *fully parenthesised* if it is either a single matrix, or a product of two fully parenthesised matrix products, surrounded by parenthesis.

Matrix Chain Multiplication problem: Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices where for $i = 1,2,\ldots,n$, matrix A_i has dimension $p_{i-1} \times p_i$, find a full parenthesisation of the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ that minimises the number of scalar multiplications.

Full Parenthesisation

A product of matrices is *fully parenthesised* if it is either a single matrix, or a product of two fully parenthesised matrix products, surrounded by parenthesis.

Matrix Chain Multiplication problem: Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices where for $i = 1,2,\ldots,n$, matrix A_i has dimension $p_{i-1} \times p_i$, find a full parenthesisation of the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ that minimises the number of scalar multiplications.

We can think of the input as a sequence of dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$.

A first attempt: brute force

A first attempt: brute force

Can we perhaps try every parenthesisation possible?

A first attempt: brute force

Can we perhaps try every parenthesisation possible?

For $n = 1$, we only have one matrix, so only one parenthesisation.

A first attempt: brute force

Can we perhaps try every parenthesisation possible?

For $n = 1$, we only have one matrix, so only one parenthesisation.

For $n \geq 2$, a fully parenthesised product is the product of two fully parenthesised matrix sub-products.

A first attempt: brute force

Can we perhaps try every parenthesisation possible?

For $n = 1$, we only have one matrix, so only one parenthesisation.

For $n \geq 2$, a fully parenthesised product is the product of two fully parenthesised matrix sub-products.

The split between those sub products happens between the k-th and the $(k+1)$-th matrices, for any $k = 1,\ldots,n-1$

A first attempt: brute force

Can we perhaps try every parenthesisation possible?

For $n = 1$, we only have one matrix, so only one parenthesisation.

For $n \geq 2$, a fully parenthesised product is the product of two fully parenthesised matrix sub-products.

The split between those sub products happens between the k-th and the $(k+1)$-th matrices, for any $k = 1, \ldots, n-1$

The running time is

$P(n) = 1$, if $n = 1$

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

A first attempt: brute force

Can we perhaps try every parenthesisation possible?

For $n = 1$, we only have one matrix, so only one parenthesisation.

For $n \geq 2$, a fully parenthesised product is the product of two fully parenthesised matrix sub-products.

The split between those sub products happens between the k-th and the $(k+1)$-th matrices, for any $k = 1, \ldots, n-1$

The running time is

$P(n) = 1$, if $n = 1$

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

This recurrence relation evaluates to $\Omega(2^n)$

Dynamic Programming

Dynamic Programming

Notation: Let $A_{i:j}$, where $i \leq j$, denote the matrix that results from evaluating the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$.

Dynamic Programming

Notation: Let $A_{i:j}$, where $i \leq j$, denote the matrix that results from evaluating the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$.

If $i = j$, then the parenthesisation problem for this product is trivial, as we only have one matrix.

Dynamic Programming

Notation: Let $A_{i:j}$, where $i \leq j$, denote the matrix that results from evaluating the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$.

If $i = j$, then the parenthesisation problem for this product is trivial, as we only have one matrix.

If $i < j$, then there is some $k \in [i, j)$ such that the product is split between A_k and A_{k+1}.

Dynamic Programming

Notation: Let $A_{i:j}$, where $i \leq j$, denote the matrix that results from evaluating the product $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$.

If $i = j$, then the parenthesisation problem for this product is trivial, as we only have one matrix.

If $i < j$, then there is some $k \in [i, j)$ such that the product is split between A_k and A_{k+1}.

In other words, there is a k for which the product $A_{i:j}$ can be written as the product of $A_{i:k}$ and $A_{k+1:j}$.

Optimal Substructure

In other words, there is a k for which the product $A_{i:j}$ can be written as the product of $A_{i:k}$ and $A_{k+1:j}$.

Optimal Substructure

In other words, there is a k for which the product $A_{i:j}$ can be written as the product of $A_{i:k}$ and $A_{k+1:j}$.

Consider the way to parenthesise the "prefix" $A_i \cdot \ldots \cdot A_k$.

Optimal Substructure

In other words, there is a k for which the product $A_{i:j}$ can be written as the product of $A_{i:k}$ and $A_{k+1:j}$.

Consider the way to parenthesise the “prefix” $A_i \cdot \ldots \cdot A_k$.

Within the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ there must be an optimal parenthesisation of $A_i \cdot \ldots \cdot A_k$.

Optimal Substructure

In other words, there is a k for which the product $A_{i:j}$ can be written as the product of $A_{i:k}$ and $A_{k+1:j}$.

Consider the way to parenthesise the "prefix" $A_i \cdot \ldots \cdot A_k$.

Within the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ there must be an optimal parenthesisation of $A_i \cdot \ldots \cdot A_k$.

If not, there is a cheaper parenthesisation of $A_i \cdot \ldots \cdot A_k$. We can use that one in the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ instead to obtain an overall lower cost, *a contradiction*.

Optimal Substructure

In other words, there is a k for which the product $A_{i:j}$ can be written as the product of $A_{i:k}$ and $A_{k+1:j}$.

Consider the way to parenthesise the "prefix" $A_i \cdot \ldots \cdot A_k$.

Within the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ there must be an optimal parenthesisation of $A_i \cdot \ldots \cdot A_k$.

If not, there is a cheaper parenthesisation of $A_i \cdot \ldots \cdot A_k$. We can use that one in the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ instead to obtain an overall lower cost, *a contradiction*.

Similarly for the the way to parenthesise the "suffix" $A_{k+1} \cdot \ldots \cdot A_j$.

Optimal Substructure

Optimal Substructure

For any $i < j$, the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$, and computes optimal parenthesisations for the two subproducts.

Optimal Substructure

For any $i < j$, the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$, and computes optimal parenthesisations for the two subproducts.

$$A_i \cdot A_{i+1} \cdot \ldots \cdot A_{\ell-1}, A_\ell, A_{\ell+1} \ldots \cdot A_{j-1} \cdot A_j$$

Optimal Substructure

For any $i < j$, the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$, and computes optimal parenthesisations for the two subproducts.

$$A_i \cdot A_{i+1} \cdot \ldots \cdot A_{\ell-1}, A_\ell, A_{\ell+1} \ldots \cdot A_{j-1} \cdot A_j$$

Optimal Substructure

For any $i < j$, the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$, and computes optimal parenthesisations for the two subproducts.

$$A_i \cdot A_{i+1} \cdot \ldots \cdot A_{\ell-1}, A_\ell, A_{\ell+1} \ldots \cdot A_{j-1} \cdot A_j$$

Optimal Substructure

For any $i < j$, the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$, and computes optimal parenthesisations for the two subproducts.

$$A_i \cdot A_{i+1} \cdot \ldots \cdot A_{\ell-1}, A_\ell, A_{\ell+1} \ldots \cdot A_{j-1} \cdot A_j$$

Optimal Substructure

For any $i < j$, the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$, and computes optimal parenthesisations for the two subproducts.

$$A_i \cdot A_{i+1} \cdot \ldots \cdot A_{\ell-1}, A_\ell, A_{\ell+1} \ldots \cdot A_{j-1} \cdot A_j$$

Optimal Substructure

For any $i < j$, the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$, and computes optimal parenthesisations for the two subproducts.

$$A_i \cdot A_{i+1} \cdot \ldots \cdot A_{\ell-1}, A_\ell, A_{\ell+1} \ldots \cdot A_{j-1} \cdot A_j$$

Optimal Substructure

For any $i < j$, the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$, and computes optimal parenthesisations for the two subproducts.

$$A_i \cdot A_{i+1} \cdot \ldots \cdot A_{\ell-1}, A_\ell, A_{\ell+1} \ldots \cdot A_{j-1} \cdot A_j$$

Optimal Substructure

For any $i < j$, the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$, and computes optimal parenthesisations for the two subproducts.

$$A_i \cdot A_{i+1} \cdot \ldots \cdot A_{\ell-1}, A_\ell, A_{\ell+1} \ldots \cdot A_{j-1} \cdot A_j$$

We must consider all the possible splits, i.e., choices of k.

Defining a recursion

Defining a recursion

“The optimal parenthesisation chooses a split point and the computes optimal parenthesisations for the subproblems on the left and on the right.”

Defining a recursion

“The optimal parenthesisation chooses a split point and the computes optimal parenthesisations for the subproblems on the left and on the right.”

Sounds like recursion!

Defining a recursion

"The optimal parenthesisation chooses a split point and the computes optimal parenthesisations for the subproblems on the left and on the right."

Sounds like recursion!

Let $M[i, j]$ denote the minimum cost (the minimum number of scalar multiplications) required to compute the matrix $A_{i:j}$.

Defining a recursion

"The optimal parenthesisation chooses a split point and the computes optimal parenthesisations for the subproblems on the left and on the right."

Sounds like recursion!

Let $M[i, j]$ denote the minimum cost (the minimum number of scalar multiplications) required to compute the matrix $A_{i:j}$.

What is the minimum cost of our problem?

Defining a recursion

"The optimal parenthesisation chooses a split point and the computes optimal parenthesisations for the subproblems on the left and on the right."

Sounds like recursion!

Let $M[i, j]$ denote the minimum cost (the minimum number of scalar multiplications) required to compute the matrix $A_{i:j}$.

What is the minimum cost of our problem?

$M[1,n]$.

Defining a recursion

Let $M[i, j]$ denote the minimum cost (the minimum number of scalar multiplications) required to compute the matrix $A_{i:j}$.

What is the minimum cost of our problem?

$M[1,n]$.

Defining a recursion

Let $M[i, j]$ denote the minimum cost (the minimum number of scalar multiplications) required to compute the matrix $A_{i:j}$.

What is the minimum cost of our problem?

$M[1,n]$.

What is $M[i, i]$ for any $i = 1,2,\ldots,n$?

Defining a recursion

Let $M[i, j]$ denote the minimum cost (the minimum number of scalar multiplications) required to compute the matrix $A_{i:j}$.

What is the minimum cost of our problem?

$M[1,n]$.

What is $M[i, i]$ for any $i = 1,2,\ldots,n$?

If $i = j$, we have one matrix, so 0 cost.

Defining a recursion

Let $M[i, j]$ denote the minimum cost (the minimum number of scalar multiplications) required to compute the matrix $A_{i:j}$.

What is the minimum cost of our problem?

$M[1,n]$.

What is $M[i, i]$ for any $i = 1,2,\ldots,n$?

If $i = j$, we have one matrix, so 0 cost.

$M[i, i] \ = 0$ for all $i = 1,2,\ldots,n$

Defining a recursion

Defining a recursion

Now consider the case where $i < j$.

Defining a recursion

Now consider the case where $i < j$.

Consider the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$.

Defining a recursion

Now consider the case where $i < j$.

Consider the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$.

Then $M[i, j]$ consists of

Defining a recursion

Now consider the case where $i < j$.

Consider the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$.

Then $M[i, j]$ consists of

the optimal cost of $A_i \cdot \ldots \cdot A_k$

Defining a recursion

Now consider the case where $i < j$.

Consider the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$.

Then $M[i, j]$ consists of

the optimal cost of $A_i \cdot \ldots \cdot A_k$

i.e., $M[i, k]$

Defining a recursion

Now consider the case where $i < j$.

Consider the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$.

Then $M[i, j]$ consists of

the optimal cost of $A_i \cdot \ldots \cdot A_k$

i.e., $M[i, k]$

and the optimal cost of $A_{k+1} \cdot \ldots \cdot A_j$

Defining a recursion

Now consider the case where $i < j$.

Consider the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$.

Then $M[i, j]$ consists of

the optimal cost of $A_i \cdot \ldots \cdot A_k$

i.e., $M[i, k]$

and the optimal cost of $A_{k+1} \cdot \ldots \cdot A_j$

i.e., $M[k + 1, j]$

Defining a recursion

Now consider the case where $i < j$.

Consider the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$.

Then $M[i, j]$ consists of

the optimal cost of $A_i \cdot \ldots \cdot A_k$

i.e., $M[i, k]$

and the optimal cost of $A_{k+1} \cdot \ldots \cdot A_j$

i.e., $M[k + 1, j]$

and the cost of multiplying $A_{i:k}$ and $A_{k+1:j}$

Defining a recursion

Now consider the case where $i < j$.

Consider the optimal parenthesisation of $A_i \cdot \ldots \cdot A_j$ splits the product into $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$.

Then $M[i, j]$ consists of

the optimal cost of $A_i \cdot \ldots \cdot A_k$

i.e., $M[i, k]$

and the optimal cost of $A_{k+1} \cdot \ldots \cdot A_j$

i.e., $M[k+1, j]$

and the cost of multiplying $A_{i:k}$ and $A_{k+1:j}$

i.e., $p_{i-1}p_kp_j$

Defining a recursion

We have the following relation:

Defining a recursion

We have the following relation:

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

Defining a recursion

We have the following relation:

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

Notice: $M[i,j]$ gives us the optimal costs, but not the optimal splits. To find the optimal splits, we define $S[i,j]$ to be the value k such that $M[i,j] = M[i,k] + M[k+1,j] + p_{i-1}p_kp_j$.

Computing a solution

We have the following relation:

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

We could now define a recursive algorithm straightforwardly using this.

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
          + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
          + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

Running time?

The running time is given by the following recurrence relation:

$$T(n) \;\geq \begin{cases} 1, & \text{if } n = 1, \\ 1 + \sum_{k=1}^{n-1} \left(T(k) + T(n-k) + 1\right) & \text{if } n > 1. \end{cases}$$

We could now define a recursive algorithm straightforwardly using this.

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
          + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
          + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

Running time?

The running time is given by the following recurrence relation:

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

This recurrence relation evaluates to $\Omega(2^n)$

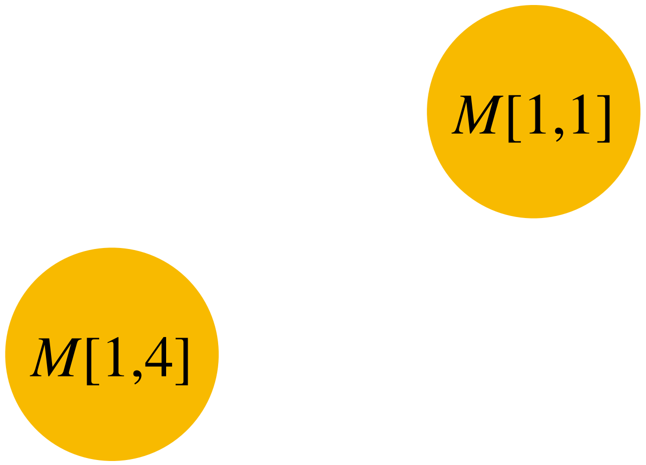
We could now define a recursive algorithm straightforwardly using this.

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
          + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
          + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

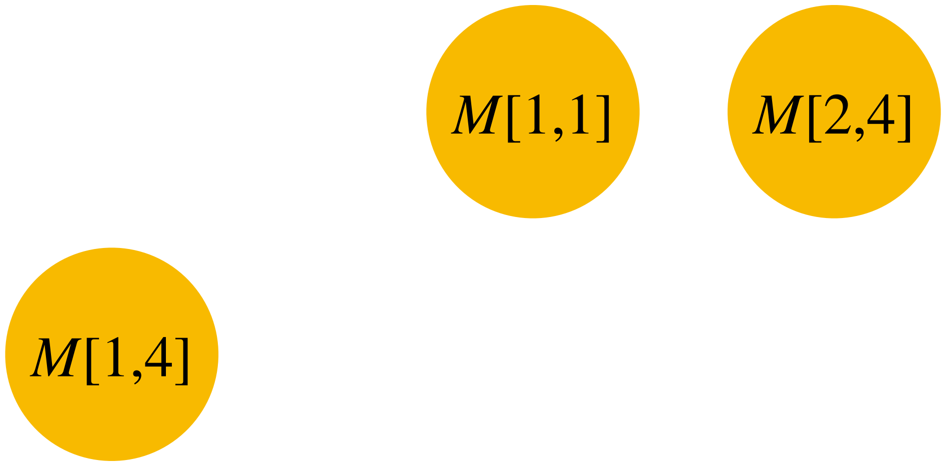
Computing the subproblems

Computing the subproblems

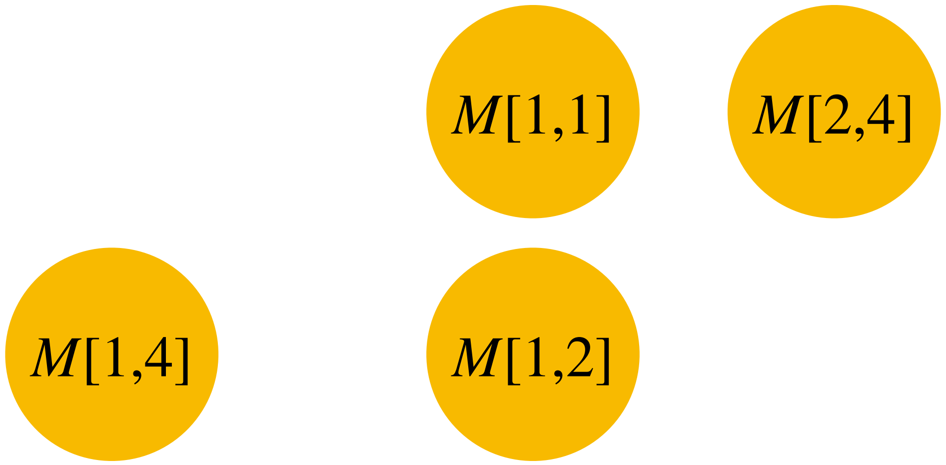
Computing the subproblems



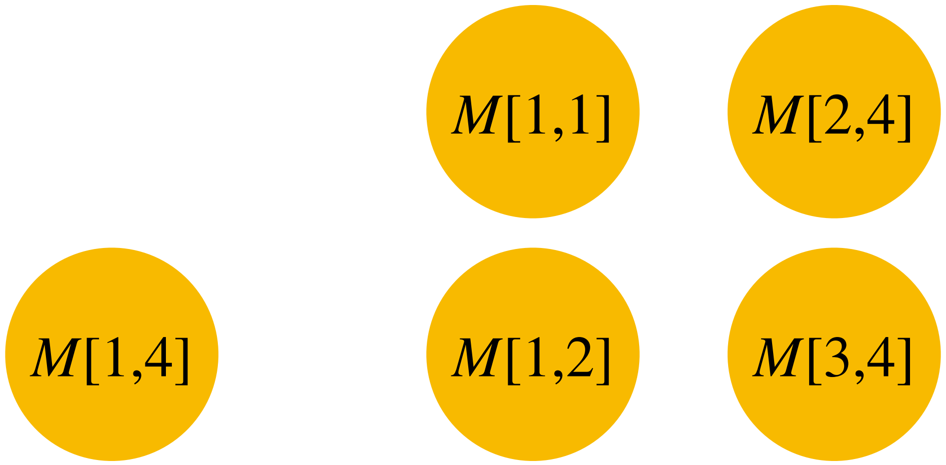
Computing the subproblems



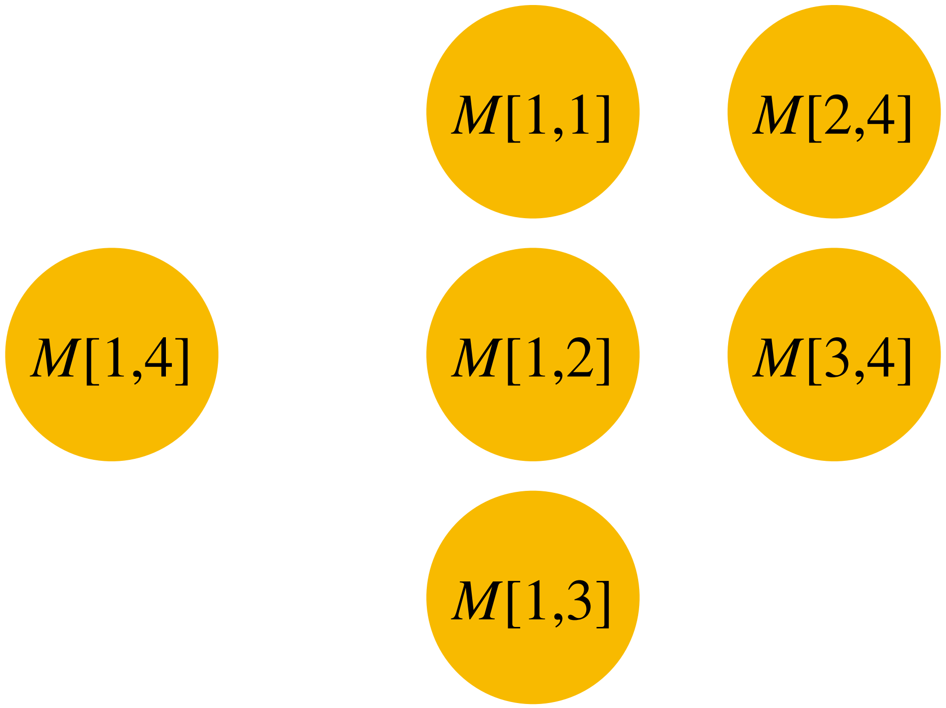
Computing the subproblems



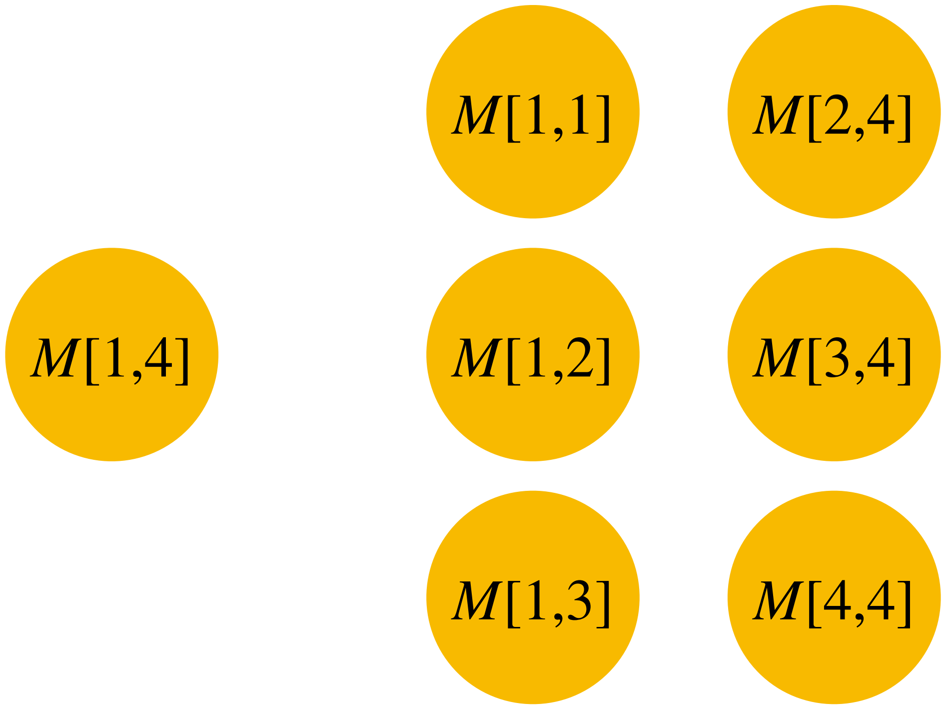
Computing the subproblems



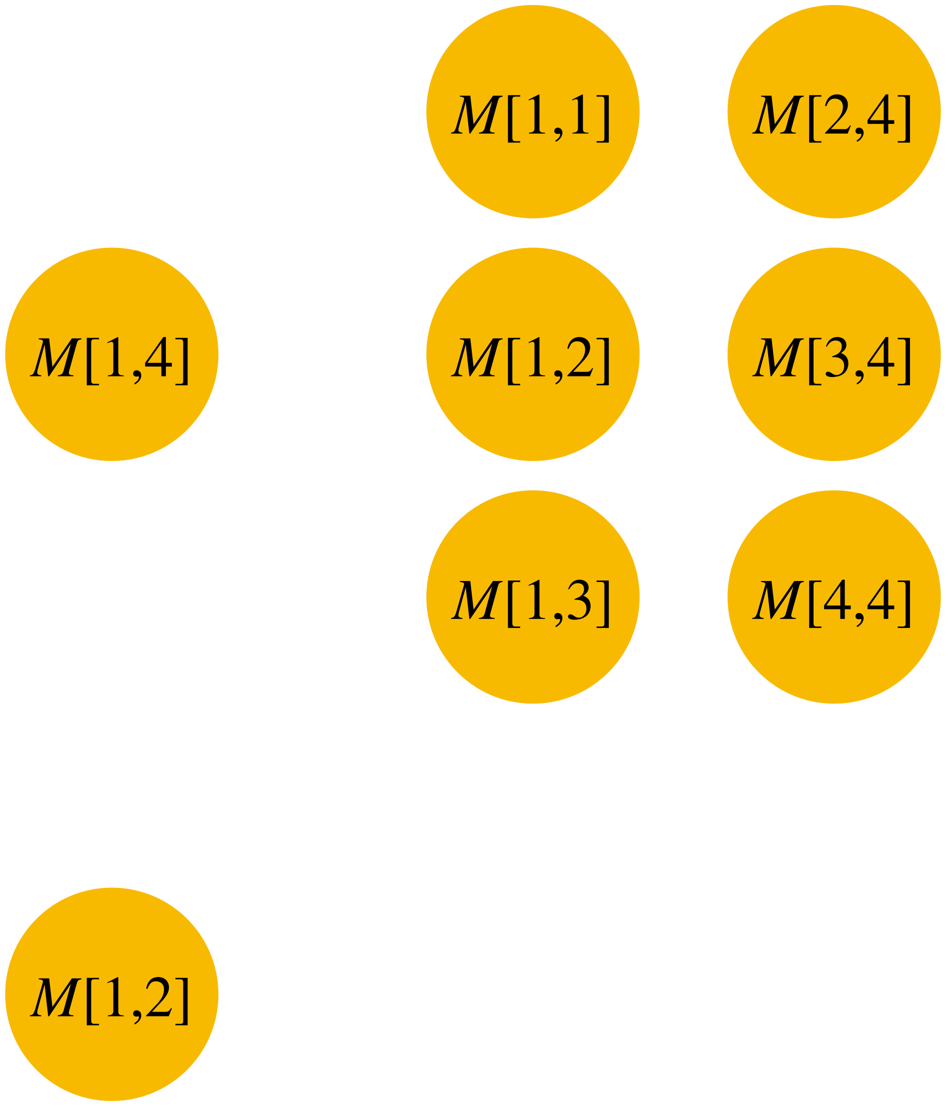
Computing the subproblems



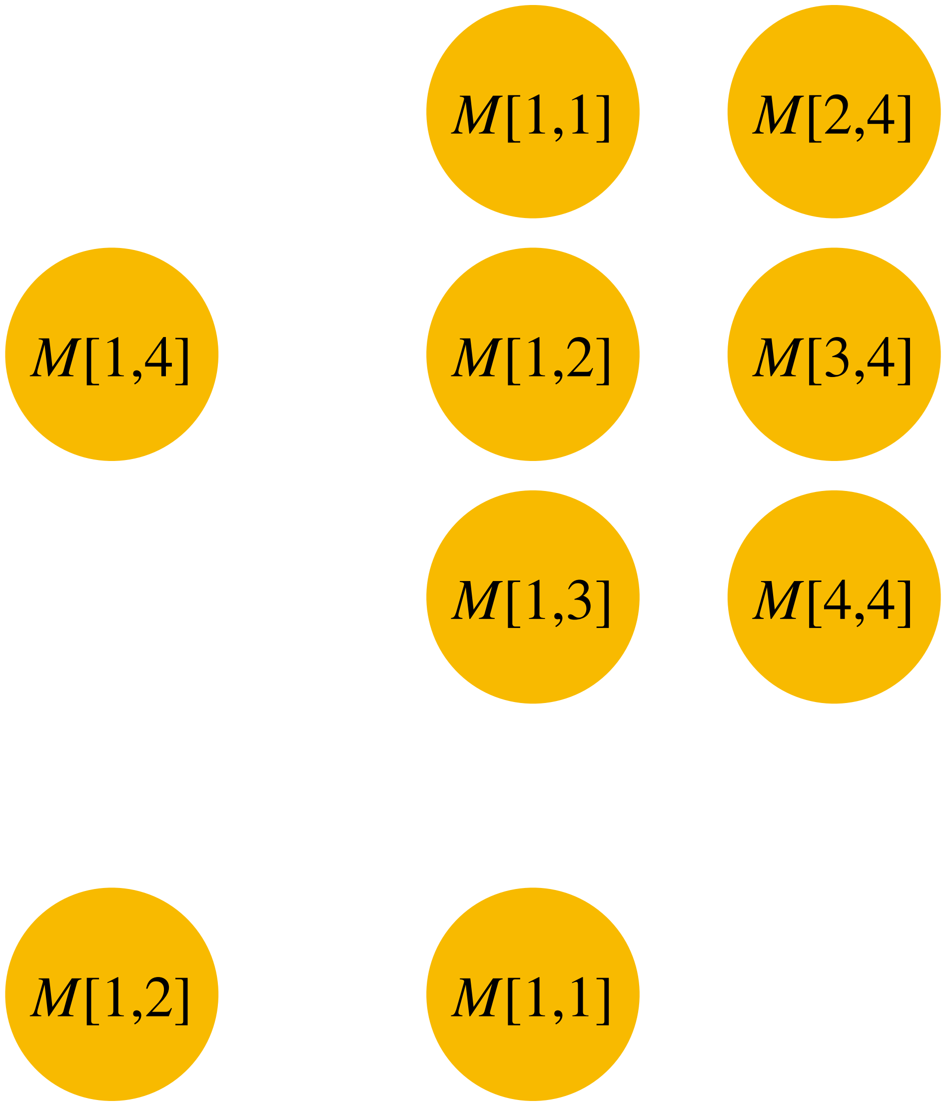
Computing the subproblems



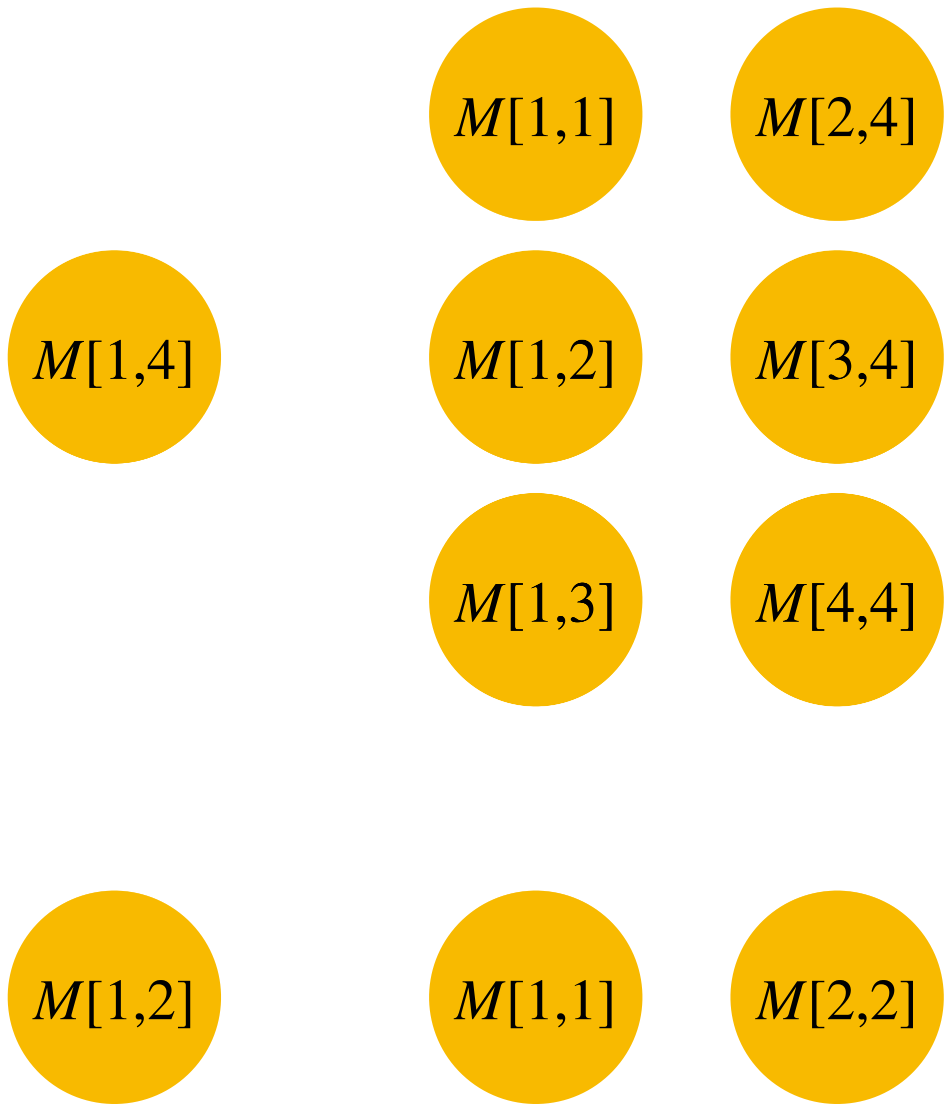
Computing the subproblems



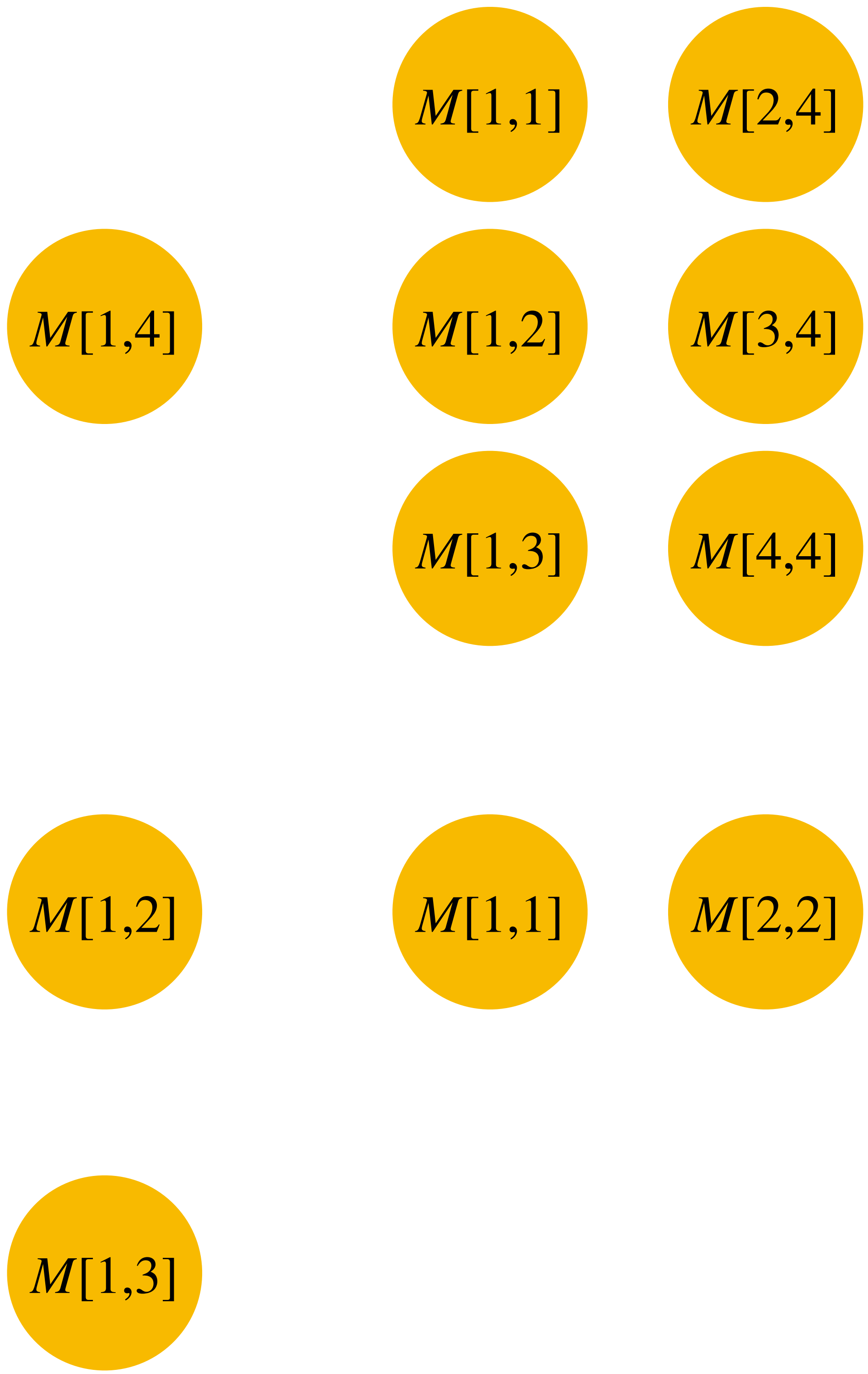
Computing the subproblems



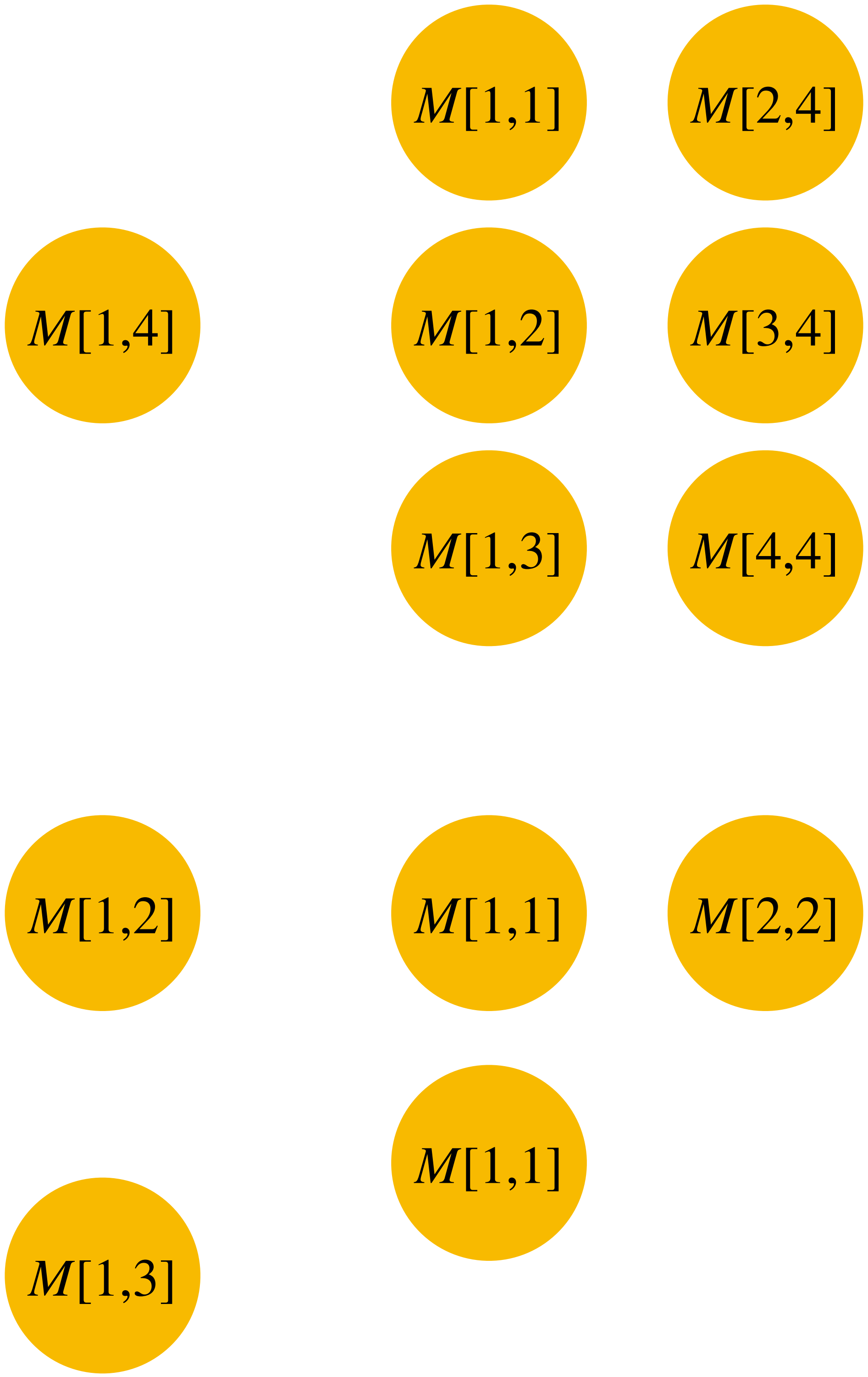
Computing the subproblems



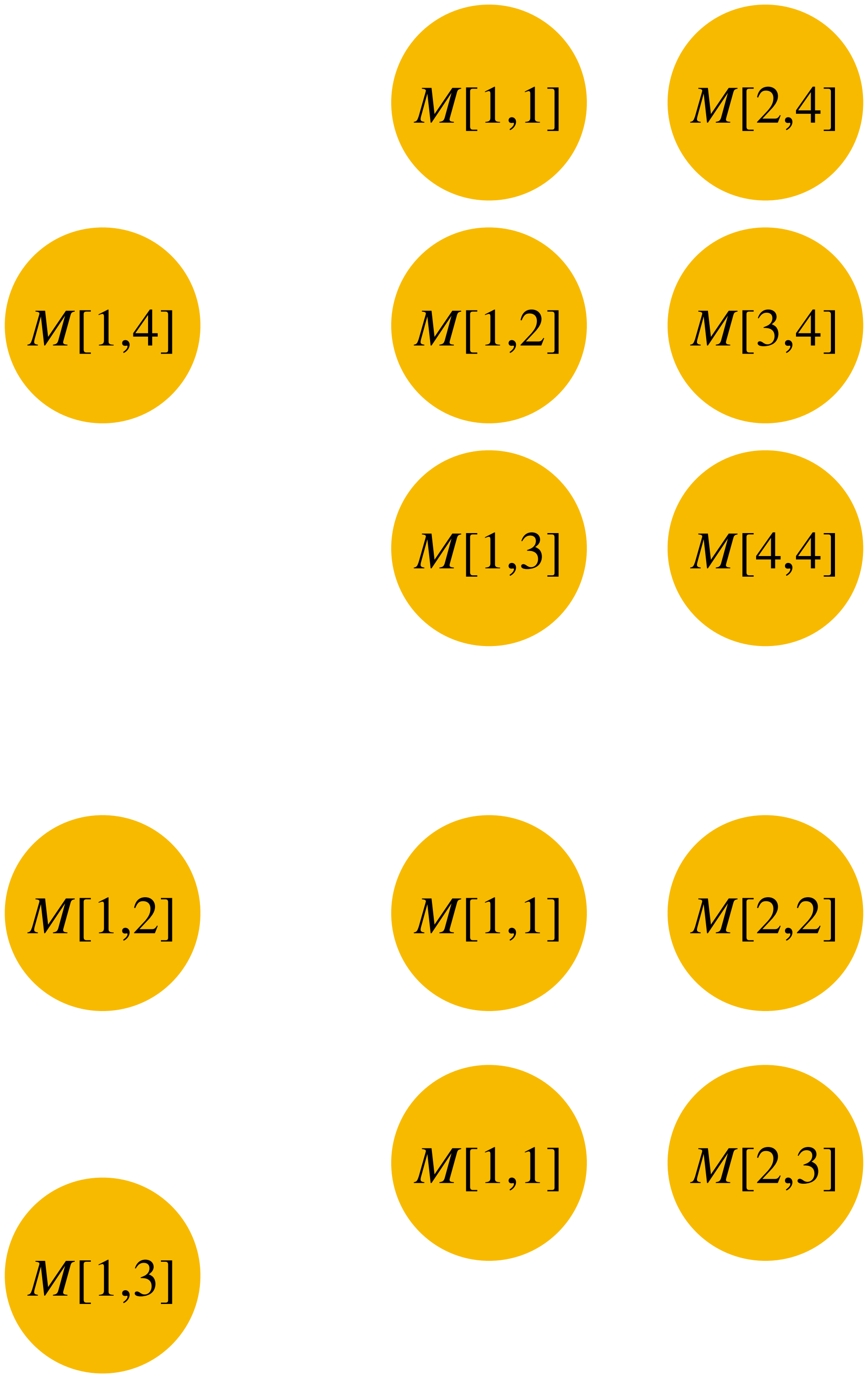
Computing the subproblems



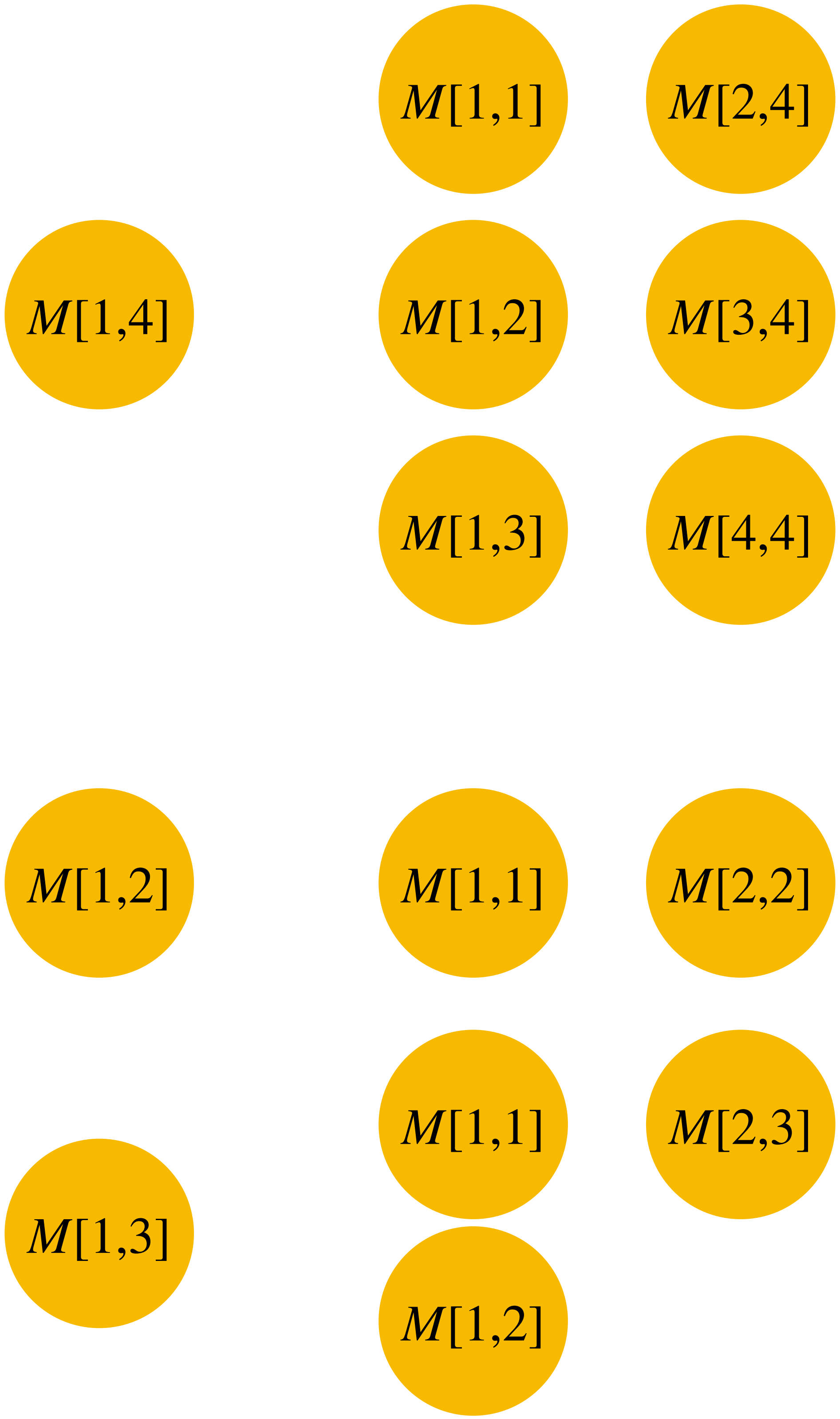
Computing the subproblems



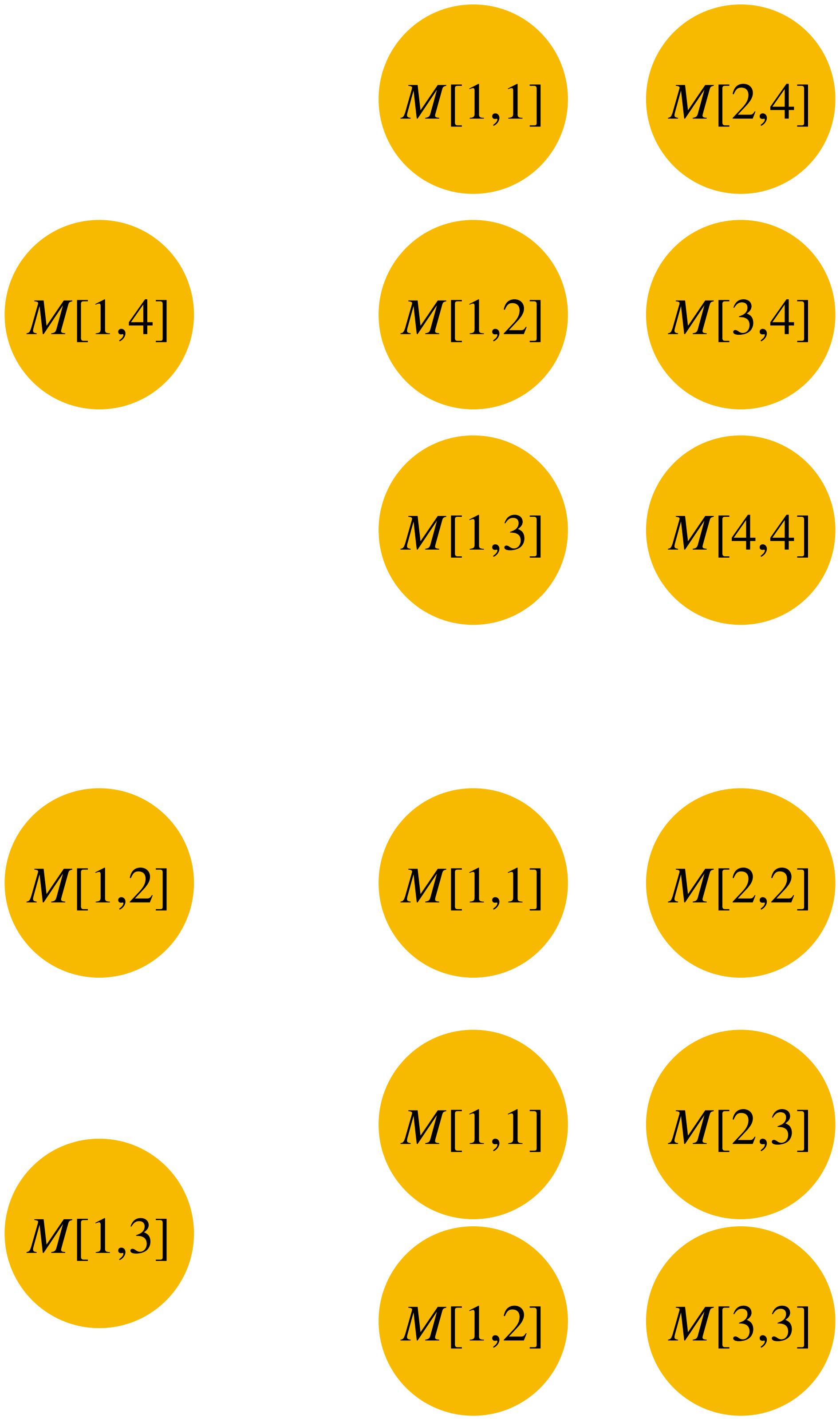
Computing the subproblems



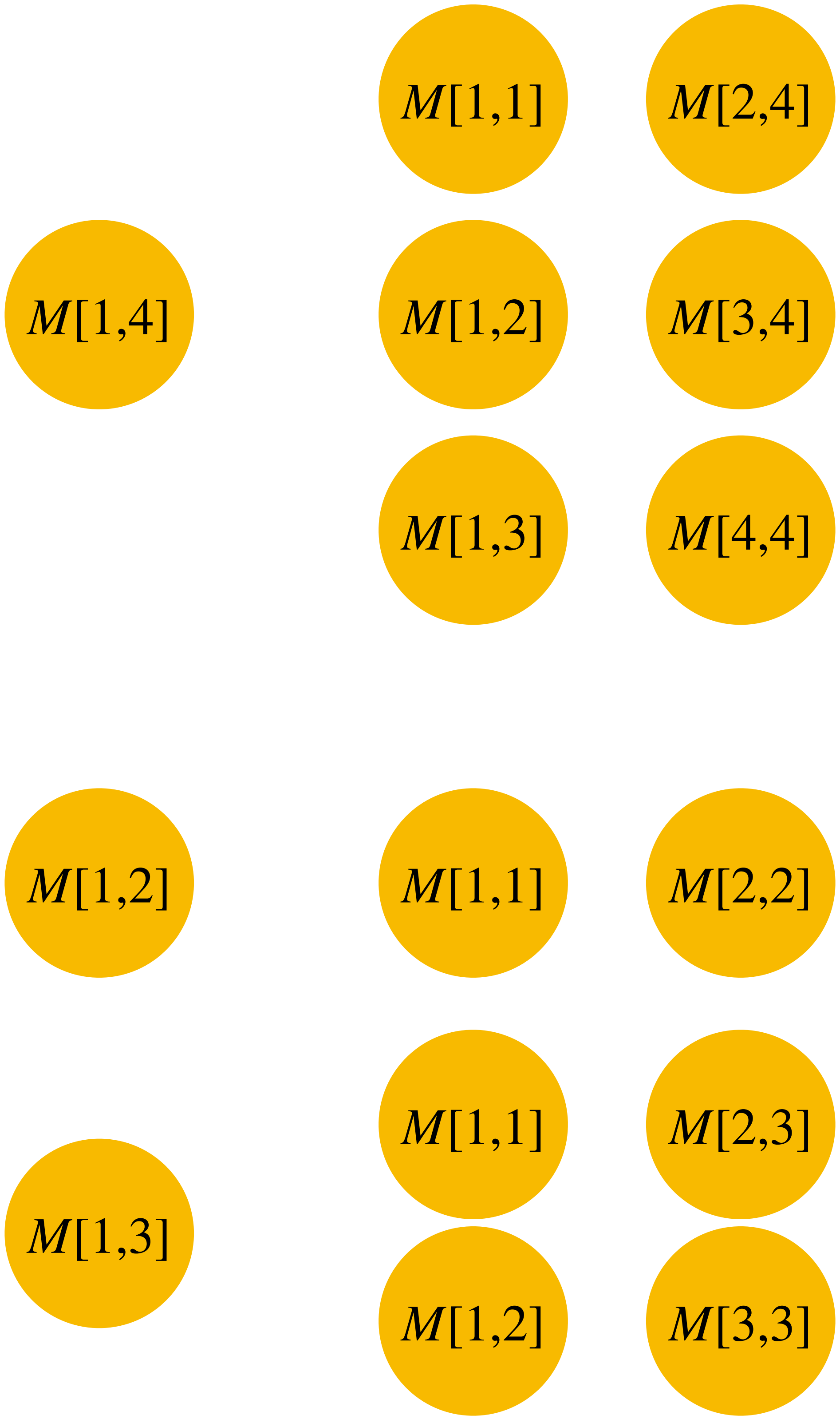
Computing the subproblems



Computing the subproblems

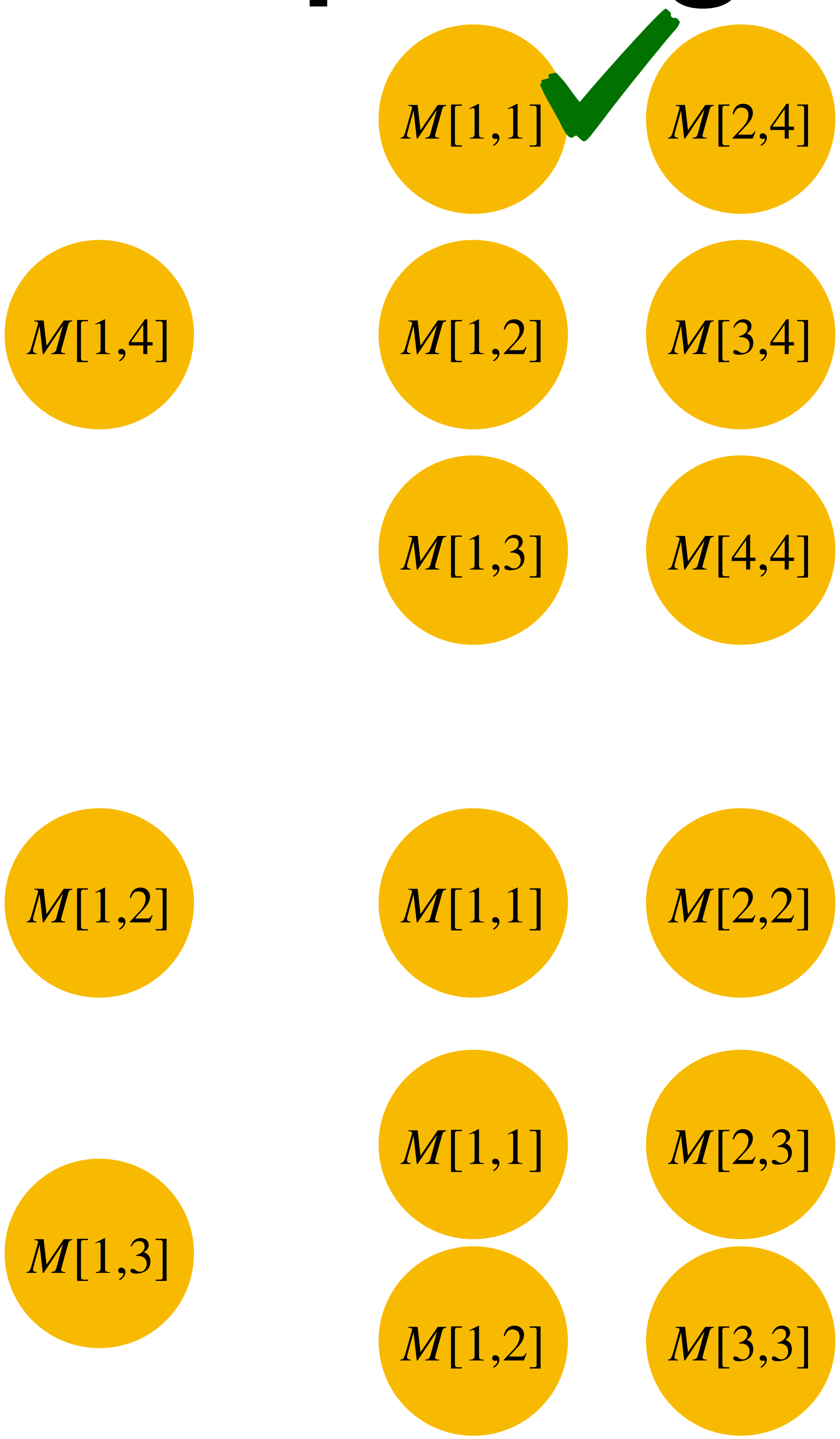


Computing the subproblems



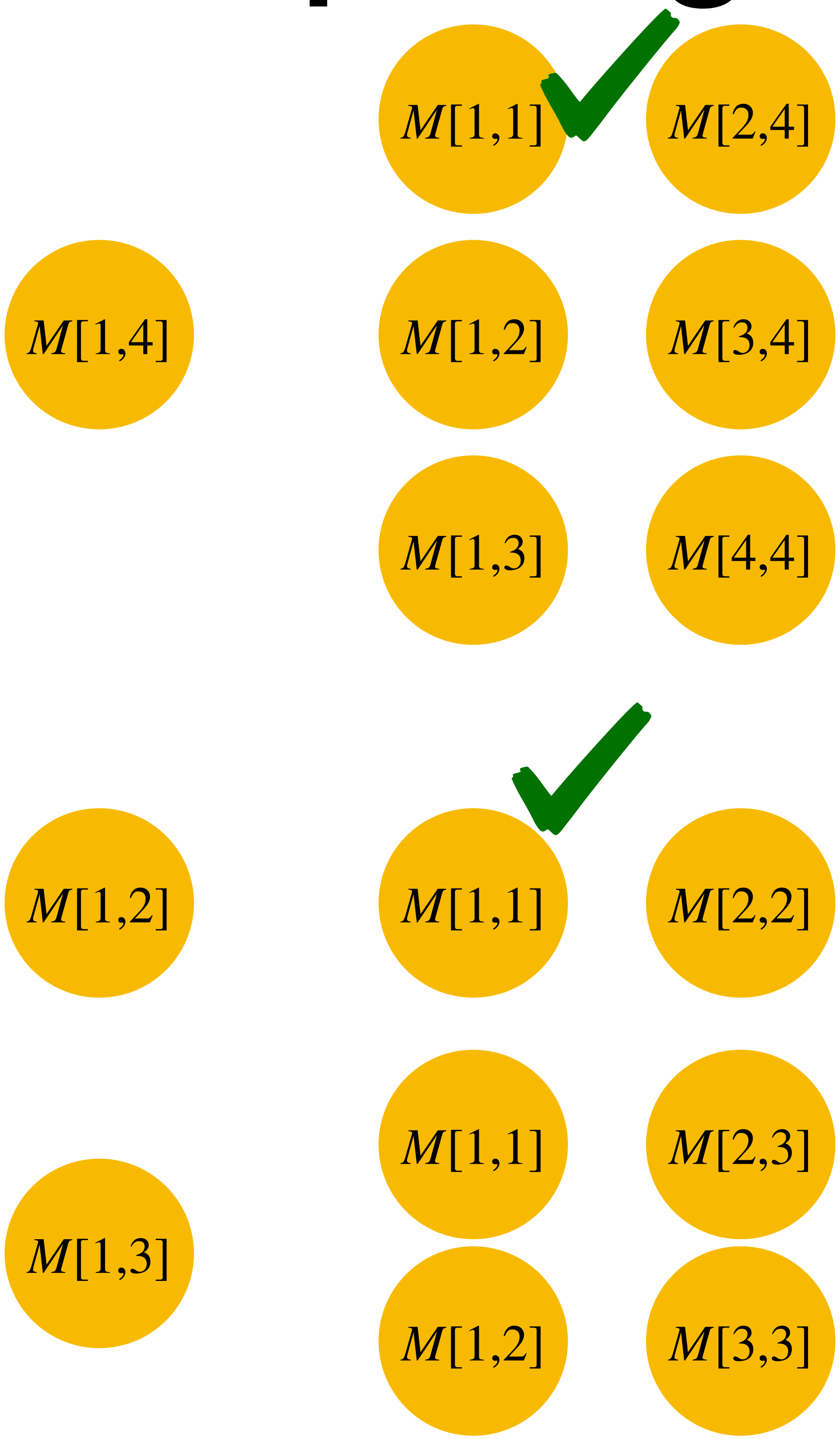
Do you observe something?

Computing the subproblems



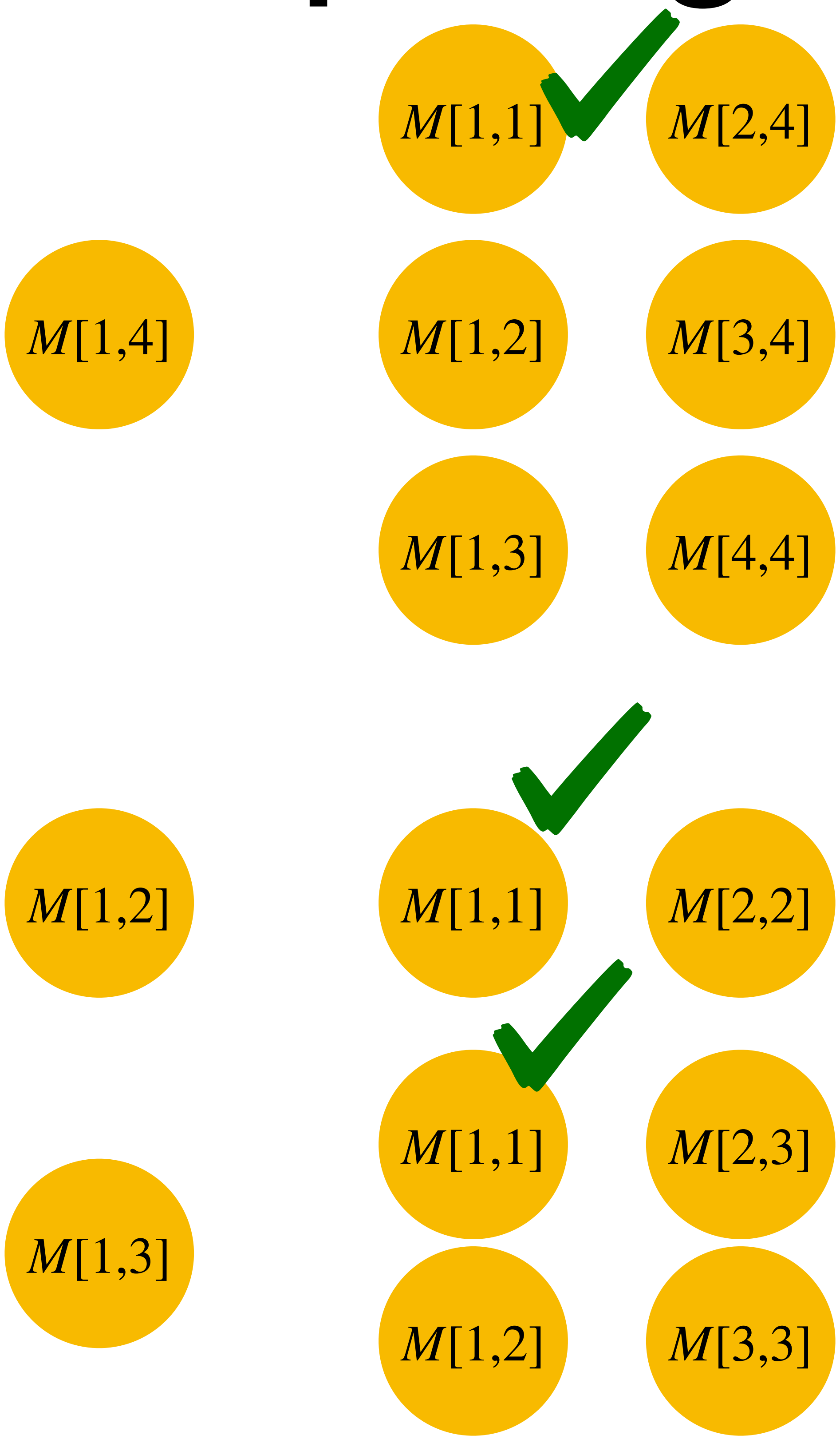
Do you observe something?

Computing the subproblems



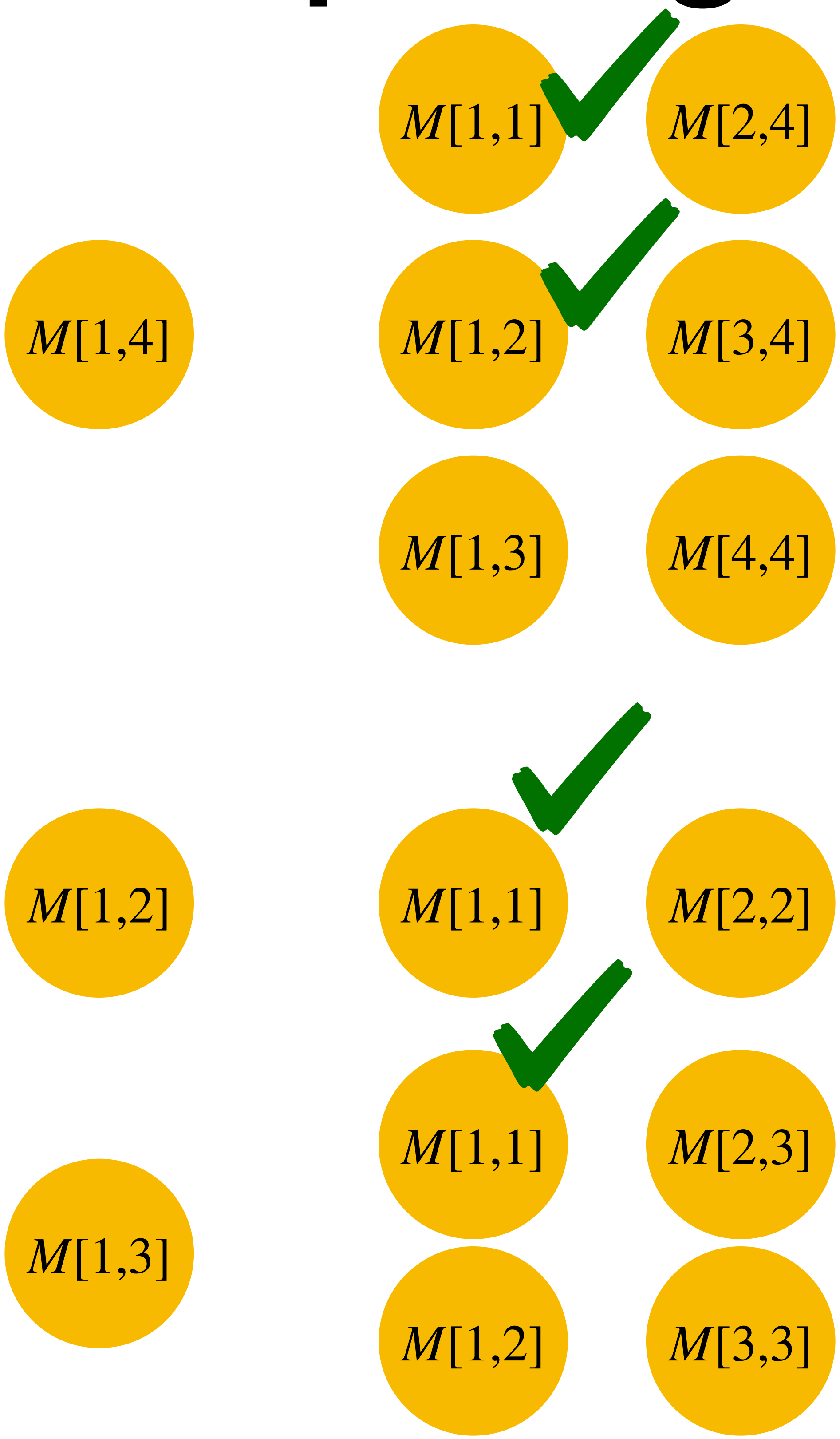
Do you observe something?

Computing the subproblems



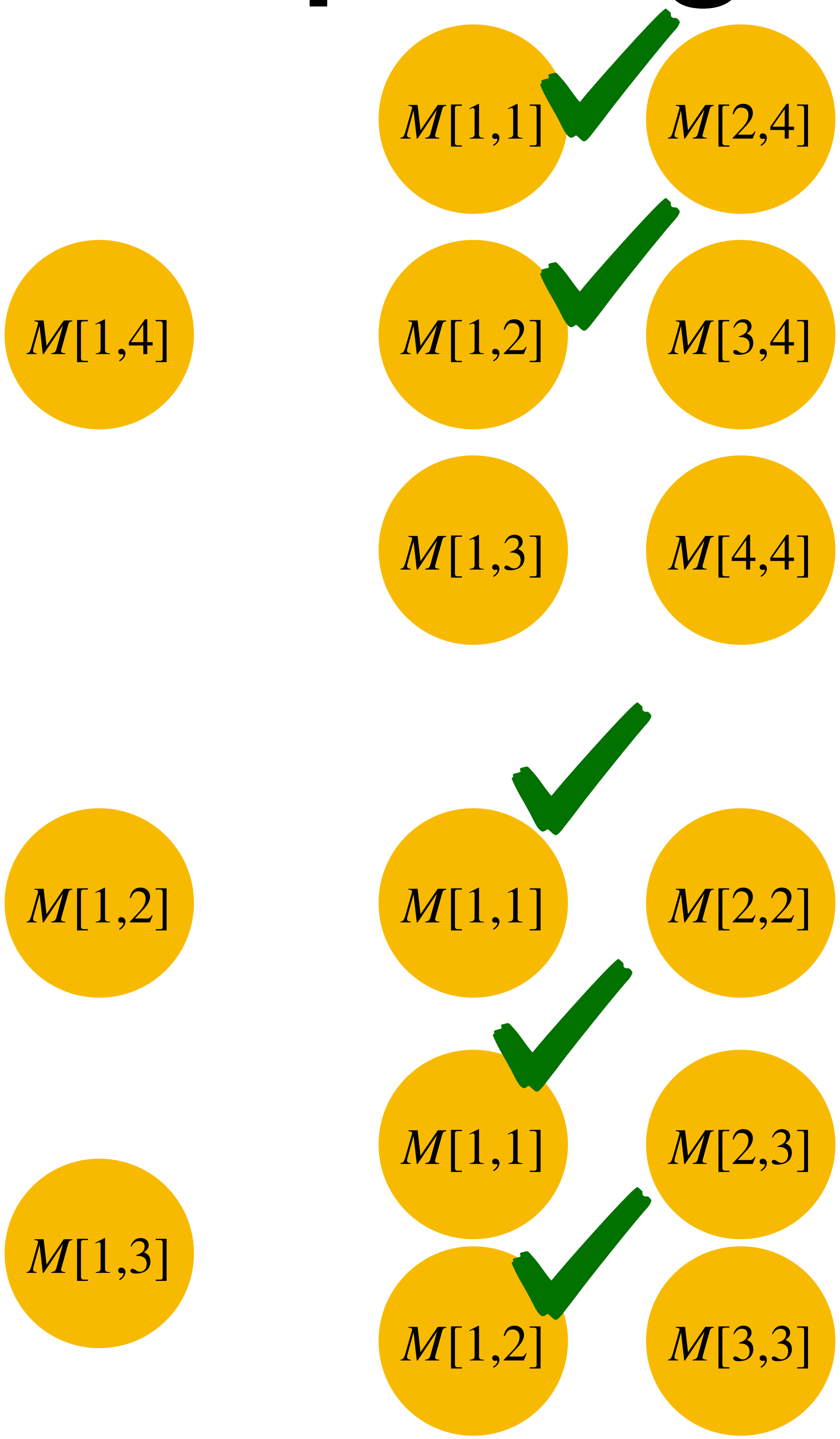
Do you observe something?

Computing the subproblems



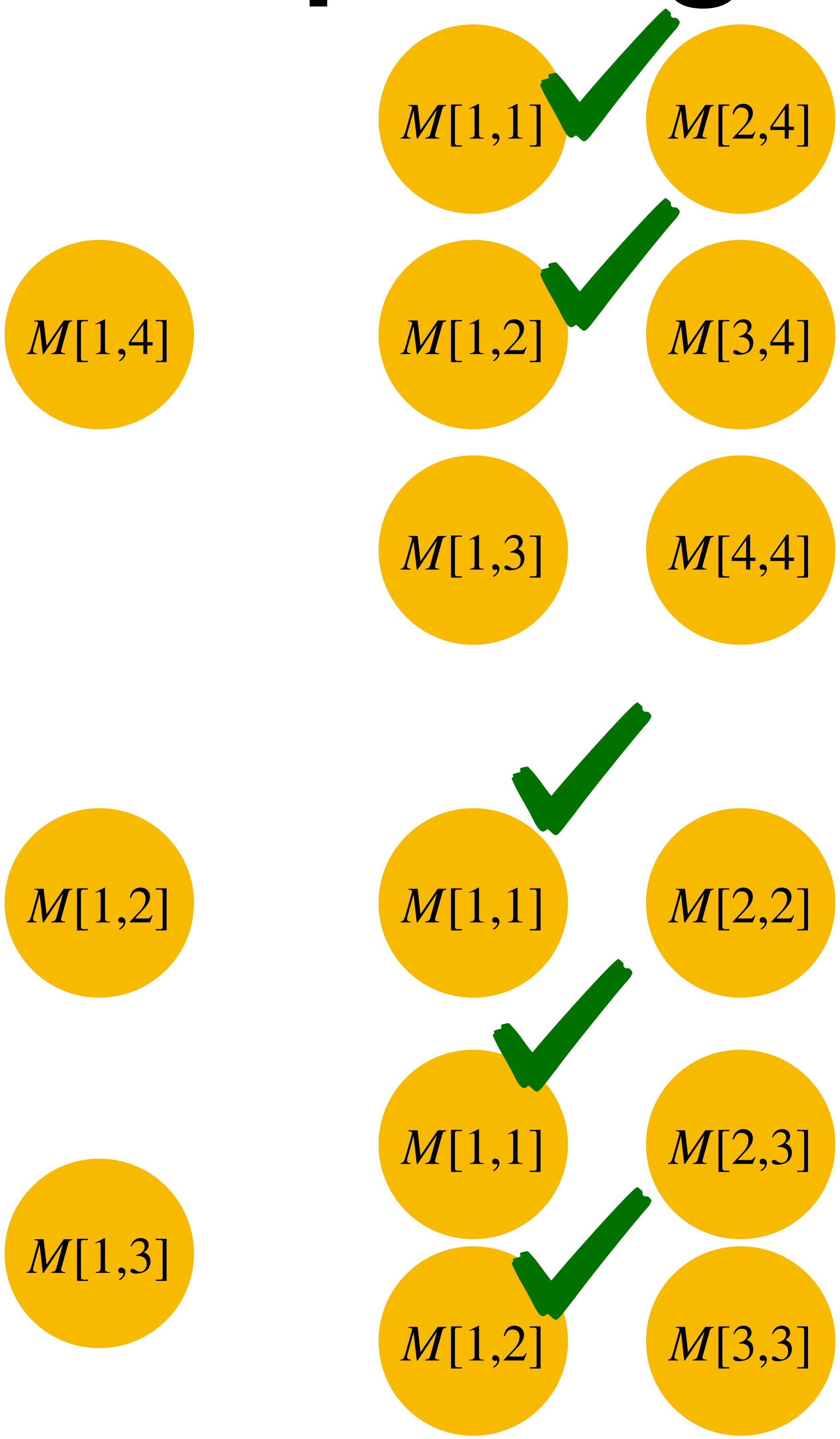
Do you observe something?

Computing the subproblems



Do you observe something?

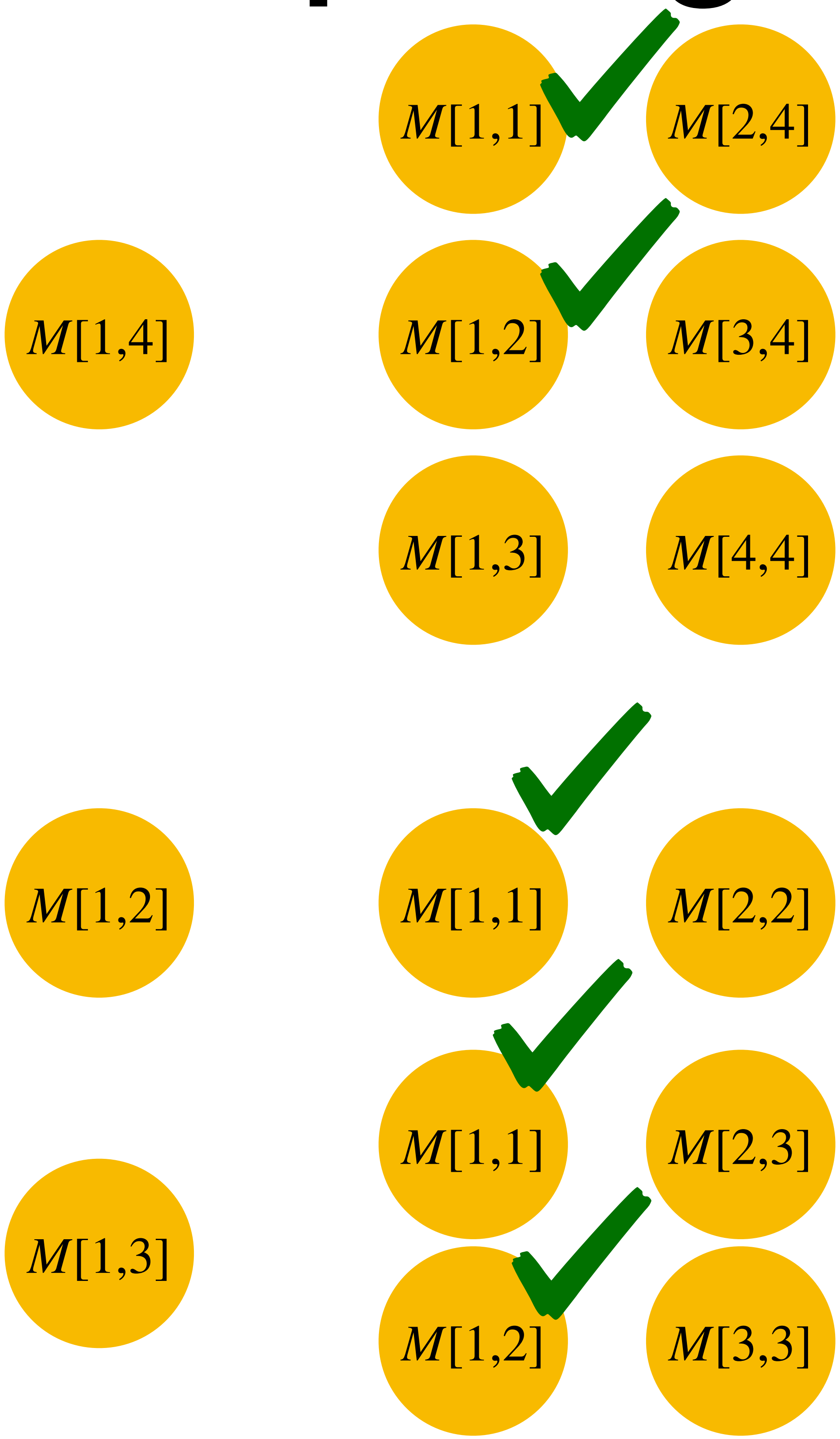
Computing the subproblems



Do you observe something?

How many distinct subproblems of the problem from i to j are there?

Computing the subproblems

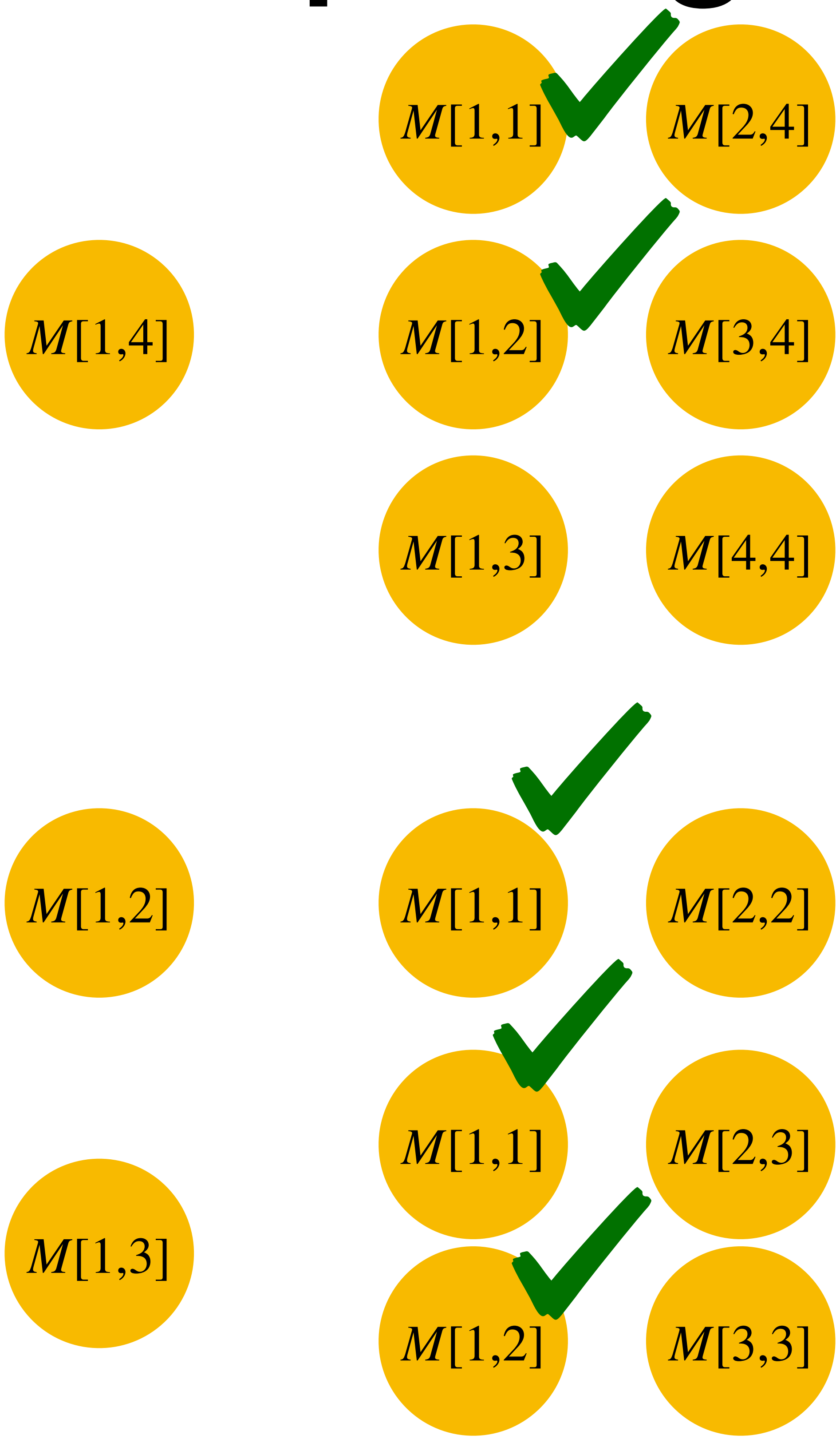


Do you observe something?

How many distinct subproblems of the problem from i to j are there?

As many as the choice of i, j that satisfy $i \leq i \leq j \leq n$, i.e., $\binom{n}{2} + n = \Theta(n^2)$

Computing the subproblems



Do you observe something?

How many distinct subproblems of the problem from i to j are there?

As many as the choice of i, j that satisfy $i \leq i \leq j \leq n$, i.e., $\binom{n}{2} + n = \Theta(n^2)$

The key is to *store* the calculation of the subproblems and reuse it.

A Dynamic Programming algorithm

A Dynamic Programming algorithm

To compute $M[i, j]$, we need the values of $M[i, k]$ and $M[k + 1, j]$ for all $i \leq k \leq j$.

A Dynamic Programming algorithm

To compute $M[i, j]$, we need the values of $M[i, k]$ and $M[k + 1, j]$ for all $i \leq k \leq j$.

So we will have to compute those first.

A Dynamic Programming algorithm

To compute $M[i, j]$, we need the values of $M[i, k]$ and $M[k+1, j]$ for all $i \leq k \leq j$.

So we will have to compute those first.

We work in a *bottom-up manner*.

A Dynamic Programming algorithm

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

Example

$A_1 : 30 \times 35$

$A_2 : 35 \times 15$

$A_3 : 15 \times 5$

$A_4 : 5 \times 10$

$A_5 : 10 \times 20$

$A_6 : 20 \times 25$

A Dynamic Programming algorithm

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

A Dynamic Programming algorithm

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

The table M

0					
	0				
		0			
			0		
				0	
					0

A Dynamic Programming algorithm

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

A Dynamic Programming algorithm

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

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER (p, n)

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

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables
for $i = 1$ **to** n // chain length 1
    $m[i,i] = 0$ } chains of length 1
for $l = 2$ **to** n chains of length 2   // l is the chain length
    for $i = 1$ **to** $n - l + 1$ // chain begins at A_i $i = 1$
        $j = i + l - 1$ // chain ends at A_j $j = 2$
        $m[i,j] = \infty$
        for $k = i$ **to** $j - 1$ // try $A_{i:k} A_{k+1:j}$
            $q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$
            if $q < m[i,j]$
                $m[i,j] = q$ // remember this cost
                $s[i,j] = k$ // remember this index
return m and s

A Dynamic Programming algorithm

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

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables
for $i = 1$ **to** n // chain length 1
    $m[i, i] = 0$
chains of length 1
for $l = 2$ **to** n chains of length 2 // l is the chain length
    for $i = 1$ **to** $n - l + 1$ // chain begins at A_i $i = 1$
        $j = i + l - 1$ // chain ends at A_j $j = 2$
        $m[i, j] = \infty$ We are now computing $M[1,2]$
        for $k = i$ **to** $j - 1$ // try $A_{i:k} A_{k+1:j}$ $k = 1$
            $q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$
            if $q < m[i, j]$
                $m[i, j] = q$ // remember this cost
                $s[i, j] = k$ // remember this index
return m and s

A Dynamic Programming algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                          // chain length 1
    m[i, i] = 0        } chains of length 1
for l = 2 to n   chains of length 2     // l is the chain length
    for i = 1 to n − l + 1              // chain begins at A_i            i = 1
        j = i + l − 1                   // chain ends at A_j              j = 2
        m[i, j] = ∞                         We are now computing M[1,2]
        for k = i to j − 1              // try A_{i:k} A_{k+1:j}          k = 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j        M[1,1] + M[2,2] + p_0 p_1 p_2
            if q < m[i, j]
                m[i, j] = q             // remember this cost
                s[i, j] = k             // remember this index
return m and s
```

A Dynamic Programming algorithm

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

$i = 1$

$j = 2$

We are now computing $M[1,2]$

$k = 1$

$M[1,1] + M[2,2] + p_0 p_1 p_2$

0

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables
for $i = 1$ **to** n // chain length 1
    $m[i,i] = 0$ chains of length 1
for $l = 2$ **to** n chains of length 2 // l is the chain length
    for $i = 1$ **to** $n - l + 1$ // chain begins at A_i $i = 1$
        $j = i + l - 1$ // chain ends at A_j $j = 2$
        $m[i,j] = \infty$ We are now computing $M[1,2]$
        for $k = i$ **to** $j - 1$ // try $A_{i:k} A_{k+1:j}$ $k = 1$
            $q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$ $\underbrace{M[1,1]}_{0} + \underbrace{M[2,2]}_{0} + p_0 p_1 p_2$
            if $q < m[i,j]$
                $m[i,j] = q$ // remember this cost
                $s[i,j] = k$ // remember this index
return m and s

Example

$A_1 : 30 \times 35$

$A_2 : 35 \times 15$

$A_3 : 15 \times 5$

$A_4 : 5 \times 10$

$A_5 : 10 \times 20$

$A_6 : 20 \times 25$

Example

$A_1 : 30 \times 35$

$p_0 p_1 p_2 = 30 \cdot 35 \cdot 15 = 15750$

$A_2 : 35 \times 15$

$A_3 : 15 \times 5$

$A_4 : 5 \times 10$

$A_5 : 10 \times 20$

$A_6 : 20 \times 25$

The table M

0	15750				
	0				
		0			
			0		
				0	
					0

A Dynamic Programming algorithm

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

A Dynamic Programming algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1:n, 1:n] and s[1:n-1, 2:n] be new tables          this goes up to 5
for i = 1 to n                                            // chain length 1
    m[i, i] = 0                      chains of length 1
for l = 2 to n    chains of length 2                      // l is the chain length
    for i = 1 to n - l + 1                                // chain begins at A_i
        j = i + l - 1                                     // chain ends at A_j
        m[i, j] = ∞
        for k = i to j - 1                                // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
            if q < m[i, j]
                m[i, j] = q                               // remember this cost
                s[i, j] = k                               // remember this index
return m and s
```

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

this goes up to 5

let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables

for $i = 1$ **to** n // chain length 1

  $m[i, i] = 0$

chains of length 1

for $l = 2$ **to** n chains of length 2 // l is the chain length

  for $i = 1$ **to** $n - l + 1$ // chain begins at A_i $i = 2$

    $j = i + l - 1$ // chain ends at A_j

    $m[i, j] = \infty$

    for $k = i$ **to** $j - 1$ // try $A_{i:k} A_{k+1:j}$

      $q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$

      if $q < m[i, j]$

        $m[i, j] = q$ // remember this cost

        $s[i, j] = k$ // remember this index

return m and s

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables　　　this goes up to 5
for $i = 1$ **to** n　　　// chain length 1
　　$m[i, i] = 0$　　　chains of length 1
for $l = 2$ **to** n　　chains of length 2　　// l is the chain length
　　for $i = 1$ **to** $n - l + 1$　　　// chain begins at A_i　　　$i = 2$
　　　　$j = i + l - 1$　　　// chain ends at A_j　　　$j = 3$
　　　　$m[i, j] = \infty$
　　　　for $k = i$ **to** $j - 1$　　　// try $A_{i:k} A_{k+1:j}$
　　　　　　$q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$
　　　　　　if $q < m[i, j]$
　　　　　　　　$m[i, j] = q$　　　// remember this cost
　　　　　　　　$s[i, j] = k$　　　// remember this index
return m and s

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

```
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables        this goes up to 5
for i = 1 to n                        // chain length 1
    m[i, i] = 0                       chains of length 1
for l = 2 to n    chains of length 2  // l is the chain length
    for i = 1 to n − l + 1            // chain begins at A_i           i = 2
        j = i + l − 1                 // chain ends at A_j             j = 3
        m[i, j] = ∞                         We are now computing M[2,3]
        for k = i to j − 1            // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q           // remember this cost
                s[i, j] = k           // remember this index
return m and s
```

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables   this goes up to 5

for $i = 1$ **to** n // chain length 1

    $m[i, i] = 0$ chains of length 1

for $l = 2$ **to** n chains of length 2   // l is the chain length

    for $i = 1$ **to** $n - l + 1$ // chain begins at A_i $i = 2$

        $j = i + l - 1$ // chain ends at A_j $j = 3$

        $m[i, j] = \infty$ We are now computing $M[2,3]$

        for $k = i$ **to** $j - 1$ // try $A_{i:k} A_{k+1:j}$ $k = 2$

            $q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$

            if $q < m[i, j]$

                $m[i, j] = q$ // remember this cost

                $s[i, j] = k$ // remember this index

return m and s

A Dynamic Programming algorithm

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

this goes up to 5

chains of length 1

chains of length 2

$i = 2$

$j = 3$

We are now computing $M[2,3]$

$k = 2$

$M[2,2] + M[3,3] + p_1 p_2 p_3$

Example

$A_1 : 30 \times 35$

$A_2 : 35 \times 15$

$A_3 : 15 \times 5$

$A_4 : 5 \times 10$

$A_5 : 10 \times 20$

$A_6 : 20 \times 25$

Example

$A_1 : 30 \times 35$

$A_2 : 35 \times 15$

$A_3 : 15 \times 5$

$A_4 : 5 \times 10$

$A_5 : 10 \times 20$

$A_6 : 20 \times 25$

$p_1 p_2 p_3 = 35 \cdot 15 \cdot 5 = 2625$

The table M

0	15750				
	0	2625			
		0			
			0		
				0	
					0

The table M

0	15750				
	0	2625			
		0	750		
			0	1000	
				0	5000
					0

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables
for $i = 1$ **to** n // chain length 1
    $m[i, i] = 0$
chains of length 1
for $l = 2$ **to** n // l is the chain length
    for $i = 1$ **to** $n - l + 1$ // chain begins at A_i
        $j = i + l - 1$ // chain ends at A_j
        $m[i, j] = \infty$
        for $k = i$ **to** $j - 1$ // try $A_{i:k} A_{k+1:j}$
            $q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$
            if $q < m[i, j]$
                $m[i, j] = q$ // remember this cost
                $s[i, j] = k$ // remember this index
return m and s

A Dynamic Programming algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                      // chain length 1
    m[i, i] = 0                     chains of length 1
for l = 2 to n   chains of length 3 // l is the chain length
    for i = 1 to n − l + 1          // chain begins at A_i
        j = i + l − 1               // chain ends at A_j
        m[i, j] = ∞
        for k = i to j − 1          // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q         // remember this cost
                s[i, j] = k         // remember this index
return m and s
```

A Dynamic Programming algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                                  // chain length 1
    m[i, i] = 0          } chains of length 1
for l = 2 to n     chains of length 3            // l is the chain length
    for i = 1 to n − l + 1                      // chain begins at A_i          i = 1
        j = i + l − 1                           // chain ends at A_j
        m[i, j] = ∞
        for k = i to j − 1                      // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q                     // remember this cost
                s[i, j] = k                     // remember this index
return m and s
```

A Dynamic Programming algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                          // chain length 1
    m[i, i] = 0        } chains of length 1
for l = 2 to n   chains of length 3     // l is the chain length
    for i = 1 to n − l + 1              // chain begins at A_i        i = 1
        j = i + l − 1                   // chain ends at A_j          j = 3
        m[i, j] = ∞
        for k = i to j − 1              // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q             // remember this cost
                s[i, j] = k             // remember this index
return m and s
```

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables
for $i = 1$ **to** n // chain length 1
    $m[i, i] = 0$ chains of length 1
for $l = 2$ **to** n chains of length 3 // l is the chain length
    for $i = 1$ **to** $n - l + 1$ // chain begins at A_i $i = 1$
        $j = i + l - 1$ // chain ends at A_j $j = 3$
        $m[i, j] = \infty$ We are now computing $M[1,3]$
        for $k = i$ **to** $j - 1$ // try $A_{i:k} A_{k+1:j}$
            $q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$
            if $q < m[i, j]$
                $m[i, j] = q$ // remember this cost
                $s[i, j] = k$ // remember this index
return m and s

A Dynamic Programming algorithm

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

chains of length 1

chains of length 3

$i = 1$

$j = 3$

We are now computing $M[1,3]$

$k = 1$

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables
for $i = 1$ **to** n // chain length 1
    $m[i,i] = 0$
chains of length 1
for $l = 2$ **to** n chains of length 3 // l is the chain length
    for $i = 1$ **to** $n - l + 1$ // chain begins at A_i $i = 1$
        $j = i + l - 1$ // chain ends at A_j $j = 3$
        $m[i,j] = \infty$ We are now computing $M[1,3]$
        for $k = i$ **to** $j - 1$ // try $A_{i:k} A_{k+1:j}$
            $q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$
            if $q < m[i,j]$
                $m[i,j] = q$ // remember this cost
                $s[i,j] = k$ // remember this index
return m and s

$k = 1$ $M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 2625 + 5250 = 7875$

A Dynamic Programming algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1:n, 1:n] and s[1:n-1, 2:n] be new tables
for i = 1 to n                               // chain length 1
    m[i, i] = 0          } chains of length 1
for l = 2 to n     chains of length 3        // l is the chain length
    for i = 1 to n - l + 1                   // chain begins at A_i          i = 1
        j = i + l - 1                        // chain ends at A_j            j = 3
        m[i, j] = ∞                                  We are now computing M[1,3]
        for k = i to j - 1                   // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
            if q < m[i, j]
                m[i, j] = q                  // remember this cost
                s[i, j] = k                  // remember this index
return m and s
```

$k = 1$ $\quad M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 2625 + 5250 = 7875$

$k = 2$

A Dynamic Programming algorithm

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

chains of length 1

chains of length 3

$i = 1$

$j = 3$

We are now computing $M[1,3]$

$k = 1$ $\quad M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 2625 + 5250 = 7875$

$k = 2$ $\quad M[1,2] + M[3,3] + p_0 p_2 p_3 = 15750 + 0 + 2250 = 18000$

The table M

0	15750	7875			
	0	2625			
		0	750		
			0	1000	
				0	5000
					0

The table M

0	15750	7875	9375		
	0	2625	4375		
		0	750	2500	
			0	1000	3500
				0	5000
					0

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

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

chains of length 1

A Dynamic Programming algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                                  // chain length 1
    m[i, i] = 0          } chains of length 1
for l = 2 to n   chains of length 4             // l is the chain length
    for i = 1 to n − l + 1                      // chain begins at A_i
        j = i + l − 1                           // chain ends at A_j
        m[i, j] = ∞
        for k = i to j − 1                      // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q                     // remember this cost
                s[i, j] = k                     // remember this index
return m and s
```

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

```
let m[1:n, 1:n] and s[1:n-1, 2:n] be new tables
for i = 1 to n                                  // chain length 1
    m[i,i] = 0          } chains of length 1
for l = 2 to n    chains of length 4            // l is the chain length
    for i = 1 to n - l + 1                      // chain begins at A_i          i = 2
        j = i + l - 1                           // chain ends at A_j
        m[i,j] = ∞
        for k = i to j - 1                      // try A_{i:k} A_{k+1:j}
            q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
            if q < m[i,j]
                m[i,j] = q                      // remember this cost
                s[i,j] = k                      // remember this index
return m and s
```

A Dynamic Programming algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                                  // chain length 1
    m[i, i] = 0          chains of length 1
for l = 2 to n   chains of length 4             // l is the chain length
    for i = 1 to n − l + 1                      // chain begins at A_i          i = 2
        j = i + l − 1                           // chain ends at A_j            j = 5
        m[i, j] = ∞
        for k = i to j − 1                      // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q                     // remember this cost
                s[i, j] = k                     // remember this index
return m and s
```

A Dynamic Programming algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1:n, 1:n] and s[1:n-1, 2:n] be new tables
for i = 1 to n                          // chain length 1
    m[i, i] = 0                         } chains of length 1
for l = 2 to n    chains of length 4    // l is the chain length
    for i = 1 to n - l + 1              // chain begins at A_i        i = 2
        j = i + l - 1                   // chain ends at A_j          j = 5
        m[i, j] = ∞                          We are now computing M[2,5]
        for k = i to j - 1              // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
            if q < m[i, j]
                m[i, j] = q             // remember this cost
                s[i, j] = k             // remember this index
return m and s
```

A Dynamic Programming algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1:n, 1:n] and s[1:n-1, 2:n] be new tables
for i = 1 to n                          // chain length 1
    m[i, i] = 0                         chains of length 1
for l = 2 to n   chains of length 4     // l is the chain length
    for i = 1 to n - l + 1              // chain begins at A_i        i = 2
        j = i + l - 1                   // chain ends at A_j          j = 5
        m[i, j] = ∞                     We are now computing M[2,5]
        for k = i to j - 1              // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
            if q < m[i, j]
                m[i, j] = q             // remember this cost
                s[i, j] = k             // remember this index
return m and s
```

$k = 2$

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

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

chains of length 1

chains of length 4

$i = 2$

$j = 5$

We are now computing $M[2,5]$

$k = 2$ $\quad M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000$

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

```
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                           // chain length 1
    m[i, i] = 0                          chains of length 1
for l = 2 to n     chains of length 4    // l is the chain length
    for i = 1 to n − l + 1               // chain begins at A_i          i = 2
        j = i + l − 1                    // chain ends at A_j            j = 5
        m[i, j] = ∞                         We are now computing M[2,5]
        for k = i to j − 1               // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q              // remember this cost
                s[i, j] = k              // remember this index
return m and s
```

$k = 2$ $\quad M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000$

$k = 3$

A Dynamic Programming algorithm

MATRIX-CHAIN-ORDER(p, n)

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

chains of length 1

chains of length 4

$i = 2$

$j = 5$

We are now computing $M[2,5]$

$k = 2 \qquad M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000$

$k = 3 \qquad M[2,3] + M[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125$

A Dynamic Programming algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                                   // chain length 1
    m[i, i] = 0            chains of length 1
for l = 2 to n    chains of length 4             // l is the chain length
    for i = 1 to n − l + 1                       // chain begins at A_i        i = 2
        j = i + l − 1                            // chain ends at A_j          j = 5
        m[i, j] = ∞                                  We are now computing M[2,5]
        for k = i to j − 1                       // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q                      // remember this cost
                s[i, j] = k                      // remember this index
return m and s
```

$k = 2$ $\quad M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000$

$k = 3$ $\quad M[2,3] + M[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125$

$k = 4$

A Dynamic Programming algorithm

```
MATRIX-CHAIN-ORDER(p, n)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                                 // chain length 1
    m[i, i] = 0         } chains of length 1
for l = 2 to n   chains of length 4            // l is the chain length
    for i = 1 to n − l + 1                     // chain begins at A_i          i = 2
        j = i + l − 1                          // chain ends at A_j            j = 5
        m[i, j] = ∞                                  We are now computing M[2,5]
        for k = i to j − 1                     // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q                    // remember this cost
                s[i, j] = k                    // remember this index
return m and s
```

$k = 2$ $\quad M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000$

$k = 3$ $\quad M[2,3] + M[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125$

$k = 4$ $\quad M[2,4] + M[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375$

The table M

0	15750	7875	9375		
	0	2625	4375	7125	
		0	750	2500	
			0	1000	3500
				0	5000
					0

The table M

0	15750	7875	9375	11875	15125
	0	2625	4375	7125	10500
		0	750	2500	5375
			0	1000	3500
				0	5000
					0

The table M

0	15750	7875	9375	11875	15125
	0	2625	4375	7125	10500
		0	750	2500	5375
			0	1000	3500
				0	5000
					0

optimal cost

Computing the optimal solution

MATRIX-CHAIN-ORDER(p, n)

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

chains of length 1

chains of length 4

$i = 2$

$j = 5$

We are now computing $M[2,5]$

$k = 2 \quad M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000$

$k = 3 \quad M[2,3] + M[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125$

$k = 4 \quad M[2,4] + M[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375$

Computing the optimal solution

```
MATRIX-CHAIN-ORDER(p, n)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                                   // chain length 1
    m[i, i] = 0          } chains of length 1
for l = 2 to n   chains of length 4              // l is the chain length
    for i = 1 to n − l + 1                       // chain begins at A_i         i = 2
        j = i + l − 1                            // chain ends at A_j           j = 5
        m[i, j] = ∞                                  We are now computing M[2,5]
        for k = i to j − 1                       // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q                      // remember this cost
                s[i, j] = k                      // remember this index
return m and s
```

$k = 2$ $\quad M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000$

$k = 3$ $\quad M[2,3] + M[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125$ optimal split

$k = 4$ $\quad M[2,4] + M[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375$

Computing the optimal solution

MATRIX-CHAIN-ORDER(p, n)

```
let m[1:n, 1:n] and s[1:n-1, 2:n] be new tables
for i = 1 to n                           // chain length 1
    m[i,i] = 0                           } chains of length 1
for l = 2 to n   chains of length 4      // l is the chain length
    for i = 1 to n - l + 1               // chain begins at A_i        i = 2
        j = i + l - 1                    // chain ends at A_j          j = 5
        m[i,j] = ∞                            We are now computing M[2,5]
        for k = i to j - 1               // try A_{i:k} A_{k+1:j}
            q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
            if q < m[i,j]
                m[i,j] = q               // remember this cost
                s[i,j] = k               // remember this index
return m and s
```

$k = 2$ $\quad M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000$

$k = 3$ $\quad M[2,3] + M[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125$ optimal split

$k = 4$ $\quad M[2,4] + M[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375$

Computing the optimal solution

```
MATRIX-CHAIN-ORDER(p, n)
let m[1:n, 1:n] and s[1:n-1, 2:n] be new tables
for i = 1 to n                                   // chain length 1
    m[i,i] = 0           } chains of length 1
for l = 2 to n    chains of length 4             // l is the chain length
    for i = 1 to n - l + 1                       // chain begins at A_i        i = 2
        j = i + l - 1                            // chain ends at A_j          j = 5
        m[i, j] = ∞                                   We are now computing M[2,5]
        for k = i to j - 1                       // try A_{i:k} A_{k+1:j}
            q = m[i,k] + m[k+1, j] + p_{i-1} p_k p_j
            if q < m[i, j]
                m[i, j] = q                      // remember this cost
                s[i, j] = k                      // remember this index     S[2,5] = 2
return m and s
```

$k = 2$ $\quad M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000$

$k = 3$ $\quad M[2,3] + M[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125$ optimal split

$k = 4$ $\quad M[2,4] + M[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375$

The table S

	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

The table S

	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

$A_{1:6} = (A_{1:3}) \cdot (A_{4:6})$

The table S

$A_{1:3} = (A_{1:1}) \cdot (A_{2:3})$

	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

$A_{1:6} = (A_{1:3}) \cdot (A_{4:6})$

The table S

$A_{1:3} = (A_{1:1}) \cdot (A_{2:3})$

	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

$A_{1:6} = (A_{1:3}) \cdot (A_{4:6})$

$A_{4:6} = (A_{4:5}) \cdot (A_{6:6})$

The table S

$A_{1:3} = (A_{1:1}) \cdot (A_{2:3})$

	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

$A_{1:6} = (A_{1:3}) \cdot (A_{4:6})$

$A_{1:6} = (A_{1:1} \cdot (A_{2:3})) \cdot ((A_{4:5}) \cdot A_{6:6})$

$A_{4:6} = (A_{4:5}) \cdot (A_{6:6})$

The table S

$A_{1:3} = (A_{1:1}) \cdot (A_{2:3})$

$A_{2:3} = (A_{2:2}) \cdot (A_{3:3})$

	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

$A_{1:6} = (A_{1:3}) \cdot (A_{4:6})$

$A_{1:6} = (A_{1:1} \cdot (A_{2:3})) \cdot ((A_{4:5}) \cdot A_{6:6})$

$A_{4:6} = (A_{4:5}) \cdot (A_{6:6})$

The table S

$A_{1:3} = (A_{1:1}) \cdot (A_{2:3})$

$A_{2:3} = (A_{2:2}) \cdot (A_{3:3})$

	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

$A_{1:6} = (A_{1:3}) \cdot (A_{4:6})$

$A_{1:6} = (A_{1:1} \cdot (A_{2:3})) \cdot ((A_{4:5}) \cdot A_{6:6})$

$A_{4:6} = (A_{4:5}) \cdot (A_{6:6})$

$A_{4:5} = (A_{4:4}) \cdot (A_{5:5})$

The table S

$A_{1:3} = (A_{1:1}) \cdot (A_{2:3})$

$A_{2:3} = (A_{2:2}) \cdot (A_{3:3})$

	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

$A_{1:6} = (A_{1:3}) \cdot (A_{4:6})$

$A_{1:6} = (A_{1:1} \cdot (A_{2:3})) \cdot ((A_{4:5}) \cdot A_{6:6})$

$A_{4:6} = (A_{4:5}) \cdot (A_{6:6})$

$A_{4:5} = (A_{4:4}) \cdot (A_{5:5})$

$$A_{1:6} = (A_{1:1} \cdot (A_{2:2} \cdot A_{3:3})) \cdot ((A_{4:4} \cdot A_{5:5}) \cdot A_{6:6}) = (A_1 \cdot (A_2 \cdot A_3)) \cdot ((A_4 \cdot A_5) \cdot A_6)$$

Computing a solution

We have the following relation:

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

We could now define a recursive algorithm straightforwardly using this.

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
          + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
          + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

Memoization

We write the procedure recursively in a natural manner, but we modify it to save the result of each subproblem in a table.

The procedure first checks if we have computed the solution already in our table.

If we have, it uses it without calling itself recursively.

If we have not, then it recurses to compute it.

A Dynamic Programming algorithm (top-down)

```
MEMOIZED-MATRIX-CHAIN(p, n)
let m[1 : n, 1 : n] be a new table
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j − 1
        q = LOOKUP-CHAIN(m, p, i, k)
              + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

A Dynamic Programming algorithm (top-down)

```
MEMOIZED-MATRIX-CHAIN(p, n)
let m[1 : n, 1 : n] be a new table
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]        Already computed? Return it!
if i == j
    m[i, j] = 0
else for k = i to j − 1
        q = LOOKUP-CHAIN(m, p, i, k)
              + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

A Dynamic Programming algorithm (top-down)

```
MEMOIZED-MATRIX-CHAIN(p, n)
let m[1 : n, 1 : n] be a new table
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j − 1
        q = LOOKUP-CHAIN(m, p, i, k)
            + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

Already computed? Return it!

If the length of the problem is 1, 0 cost

A Dynamic Programming algorithm (top-down)

```
MEMOIZED-MATRIX-CHAIN(p, n)
let m[1 : n, 1 : n] be a new table
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)
```

```
LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j − 1
        q = LOOKUP-CHAIN(m, p, i, k)
              + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

Already computed? Return it!

If the length of the problem is 1, 0 cost

Use recursion to compute q

Dynamic Programming vs Divide and Conquer

DP is an optimisation technique and is only applicable to problems with optimal substructure.

DP splits the problem into parts, finds solutions to the parts and joins them.

The parts are not significantly smaller and are overlapping.

In DP, the subproblem dependency can be represented by a DAG.

DQ is not normally used for optimisation problems.

DQ splits the problem into parts, finds solutions to the parts and joins them.

The parts are significantly smaller and do not normally overlap.

In DQ, the subproblem dependency can be represented by a tree.