Dynamic Programming: Matrix Chain Multiplication

Chapter 15.2

Dynamic Programming

- A method for solving optimization problems: problems that ask for a minimum or maximum value.
- Developing a dynamic programming solution:
  1. Characterize structure of an optimal solution
     a. Optimal substructure
     b. Overlapping subproblems
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution, avoiding overlap
  4. Construct an optimal solution from computed values.

Matrix-chain Multiplication

Given a sequence of matrices A_1, A_2, ... A_n, what is the fastest way of computing the product

$$A_1A_2 \ldots A_n$$

using the straightforward matrix multiplication algorithm?

Since matrices are associative, we can parenthesize the product in any way, each parenthesization denoting a different order of multiplication.

Full parenthesization

For example, the product of four matrices has the different parenthesizations

$(A_1\ (A_2\ (A_3\ A_4)))$
$(A_1\ ((A_2\ A_3)\ A_4))$
$((A_1\ A_2)\ (A_3\ A_4))$
$((A_1\ (A_2\ A_3))\ A_4)$
$(((A_1\ A_2)\ A_3)\ A_4)$

meaning

multiply A_3 by A_4 giving T_1
multiply A_2 by T_1 giving T_2
multiply A_1 by T_2 giving R

meaning

multiply A_2 by A_3 giving T_1
multiply A_1 by T_1 giving T_2
multiply T_2 by A_4 giving R

Matrix multiplication

We want to find the product C of two matrices A and B. A is *p* by *q*, and B is *q* by *r*.

```
MatrixMultiply(A, B) {
    allocate matrix C[1..p, 1..r]

    for i = 1 to p  {
        for j = 1 to r {
            C[i, j] = 0
            for k = 1 to q {
                C[i, j] += A[i, k] * B[k, j]
            }
        }
    }
    return C;
}
```

Matrix multiplication analysis

We use the number of multiplication operations as a proxy for the total work.

```
MatrixMultiply(A, B) {
    allocate matrix C[1..p, 1..r]

    for i = 1 to p  {
        for j = 1 to r {
            C[i, j] = 0
            for k = 1 to q {
                C[i, j] += A[i, k] * B[k, j]      // executed exactly pqr times.
            }
        }
    }
    return C;
}
```

A full analysis would leads to a bound of O(pqr), so it's a reasonable proxy.

Matrix-chain multiplication

Suppose we have a chain of 3 matrices $A_1A_2A_3$ to multiply.
Let A_1 be 10 by 100, A_2 be 100 by 5, and A_3 be 5 by 50.

If we multiply according to parenthesization $((A_1A_2)A_3)$,
we have $T_1 = A_1A_2$, costing $10 \cdot 100 \cdot 5 = 5000$ multiplications,
and $R = T_1A_3$, costing $10 \cdot 5 \cdot 50 = 2500$ multiplications,
for a total of 7500 multiplications.

If we multiply according to parenthesization $(A_1(A_2A_3))$,
we have $T_1 = A_2A_3$, costing $100 \cdot 5 \cdot 50 = 25000$ multiplications,
and $R = A_1T_1$, costing $10 \cdot 100 \cdot 50 = 50000$ multiplications,
for a total of 75000 multiplications.

The first parenthesization is 10 times faster!

Brute force

Okay, suppose we compute the cost of all parenthesizations and choose the least-cost one.

How many parenthesizations are there?

Let P(n) be the number of parenthesizations of a chain of n matrices. A parenthesization has the structure

$$((A_1A_2\ldots A_k)(A_{k+1}A_{k+2}\ldots A_n)),$$

so $P(n) = \begin{cases} 1 & if\ n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & if\ n > 1 \end{cases}$

$P(n) \in \Omega(2^n)$

Applying the methodology

1. Characterize structure of an optimal solution

An optimal solution has the structure

$$((A_1A_2\ldots A_k)(A_{k+1}A_{k+2}\ldots A_n)),$$

where $1 \le k < n$. The subproblems $A_1A_2\ldots A_k$ and $A_{k+1}A_{k+2}\ldots A_n$ in the solution must be optimal solutions (least multiplications) for those subproblems. Thus the problem has optimal substructure.

Overlapping subproblems

Suppose we consider the subproblems $A_1A_2...A_{n-1}$ and $A_2A_3...A_n$. The first of these subproblems has a possible parenthesization

$$((A_1)(A_2A_3...A_{n-1}))$$

and the second has a possible parenthesization

$$((A_2A_3...A_{n-1})A_n)$$

These two parenthesizations show that both subproblems need the solution to the same subproblem $A_2A_3...A_{n-1}$. Thus this problem has overlapping subproblems. We may apply memoization or dynamic programming.

Recursive definition of value

2. Recursively define the value of an optimal solution

Let c(i, j) be the best cost (or minimum value) for multiplying in the subproblem $A_iA_{i+1}...A_j$. Let each matrix A_i be p_{i-1} by p_i.

Then the cost of the solution to our problem is c(1, n).

There is no cost associated with a single matrix subproblem, so c(i, i) = 0.

c(i, j) (for j > i) is the minimum cost of multiplying for all breakpoints k, viz.

$$((A_iA_{i+1}...A_k)(A_{k+1}A_{k+2}...A_j)),$$

Brute force algorithm

```
solution() {
        return c(1, n);
}

c(i, j) {
        if (i=j)
            return 0;
        min = ∞;
        for k = i to j-1 {
              cost = c(i, k) + c(k+1, j) + p[i-1]*p[k]*p[j];
              if (cost < min) min = cost;
         }
         return min
}
```

$\Omega(2^n)$

Memoize!

```
solution() {
    allocate m[1..n, 1..n] = ∞;
    return c(1, n);
}

c(i, j) {
    if (m[i, j] ≠ ∞) return m[i, j];
    if (i=j) {
        m[i, j] = 0;
        return m[i, j];
    }
    …
```

```
    min = ∞;
    for k = i to j-1 {
        cost = c(i, k) + c(k+1, j) +
                p[i-1]*p[k]*p[j];
        if (cost < min) min = cost;
    }
    m[i, j] = min
    return m[i, j]
}
```

Analysis of memoization

Allocating m[] takes $O(n^2)$ or $O(1)$ time depending on model.

Consider all calls to c(i, j). Let k be the number of such calls.

Then at least $k - n^2$ of them return inside the first **if**, taking $O(1)$ time each. (Because m[] holds n^2 values and each time through the rest of the function fills in 1 previously unfilled value.)

For the remaining n^2 calls, there is $O(n)$ nonrecursive work apiece. (The recursive work is counted in the "consider all calls".)

We conclude that the total work over all calls to c(i,j) is

$$(k - n^2) \cdot O(1) + n^2 \cdot O(n)$$

$$= O(k) + O(n^3).$$

Analysis of memoization

So what is k?

solution() calls c(i, j) once.

c(i, j) passes the first **if** n^2 times, and each time this happens it has the potential to call c(i, j) 2n times.

Thus the total number of calls, k, is at most $2n^3+1$.

The total work of the algorithm is therefore the total work for solution() plus the total work in c(i,j), or

$$O(n^2) + (O(k) + O(n^3))$$
$$= O(n^2) + (O(n^3) + O(n^3))$$
$$= O(n^3).$$

That's relatively expensive, but it's a lot better than $\Omega(2^n)$.

Memoization traceback

4. Construct an optimal solution from computed values.

We can use traceback as before. Let's have traceback return a string.

```
solution() {
    allocate matrix m[1..2, 1..n] = 0
    cost = c(1, n) // fills the memo table
    return "(" + traceback(1, n) + ")" and cost;
}
```

Memoization traceback

```
traceback(i, j) {
    if(i = j) {
        return "A" + i
    }
    for k = i to j-1 {
        if(m[i, j] = m[i, k] + m[k+1, j] + p[i-1]*p[k]*p[j]) {
            return "(" + traceback(i, k) + ")(" + traceback(k+1, j) + ")"
        }
    }
}
```

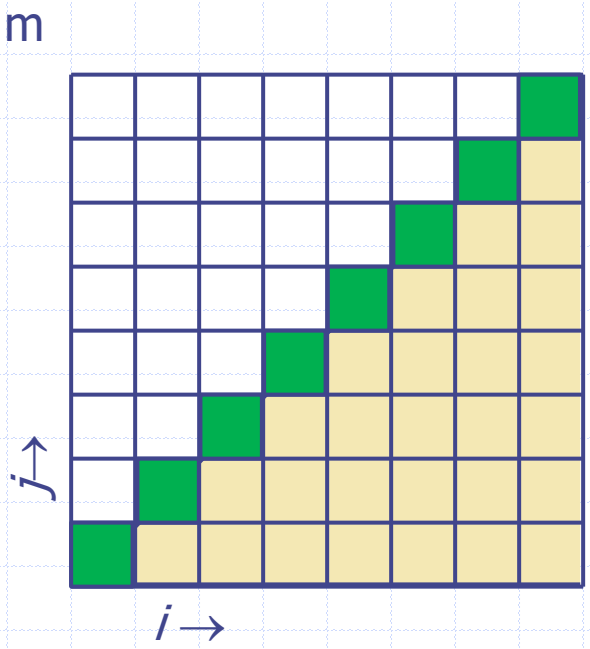
Tracing back in this fashion costs $O(n^2)$ time.

Exercise: write a modification of the code that stores the choices made, and analyze its traceback time complexity.

Dynamic Programming

Dynamic programming is computing the memos without recursion. The bottom of the recursion is when i=j so we should start with these entries.

Entries with j < i are unused.

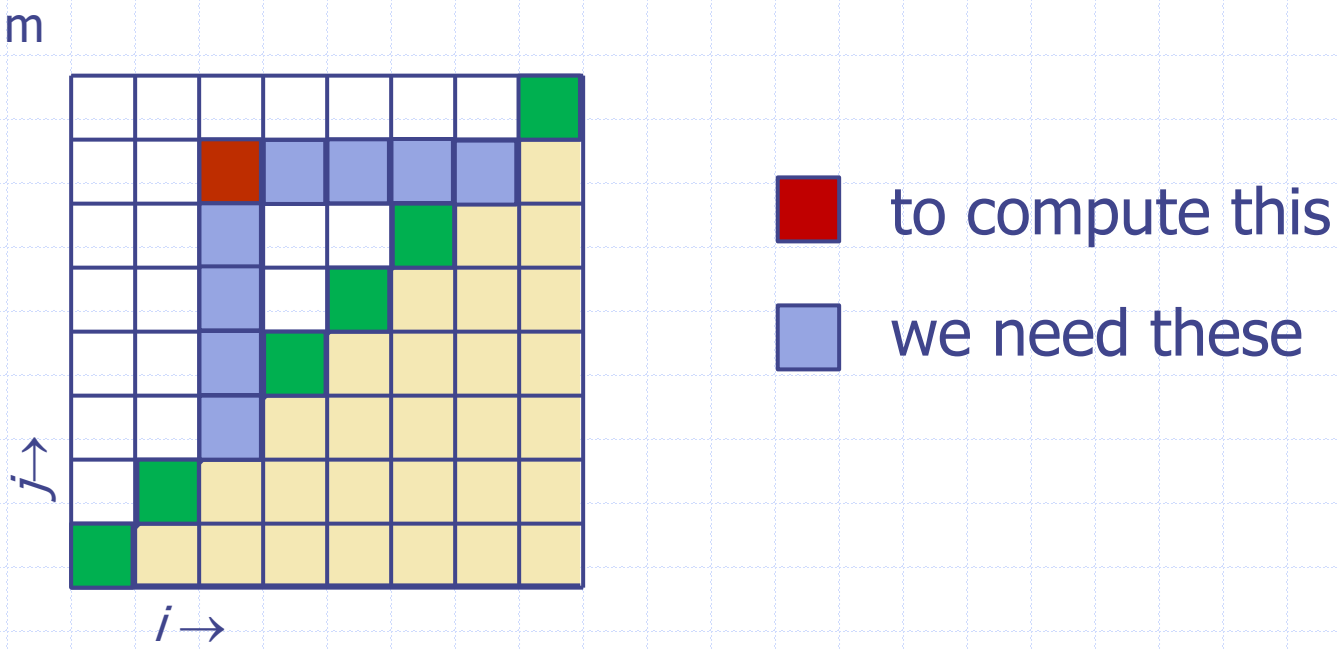


start with these

disregard these

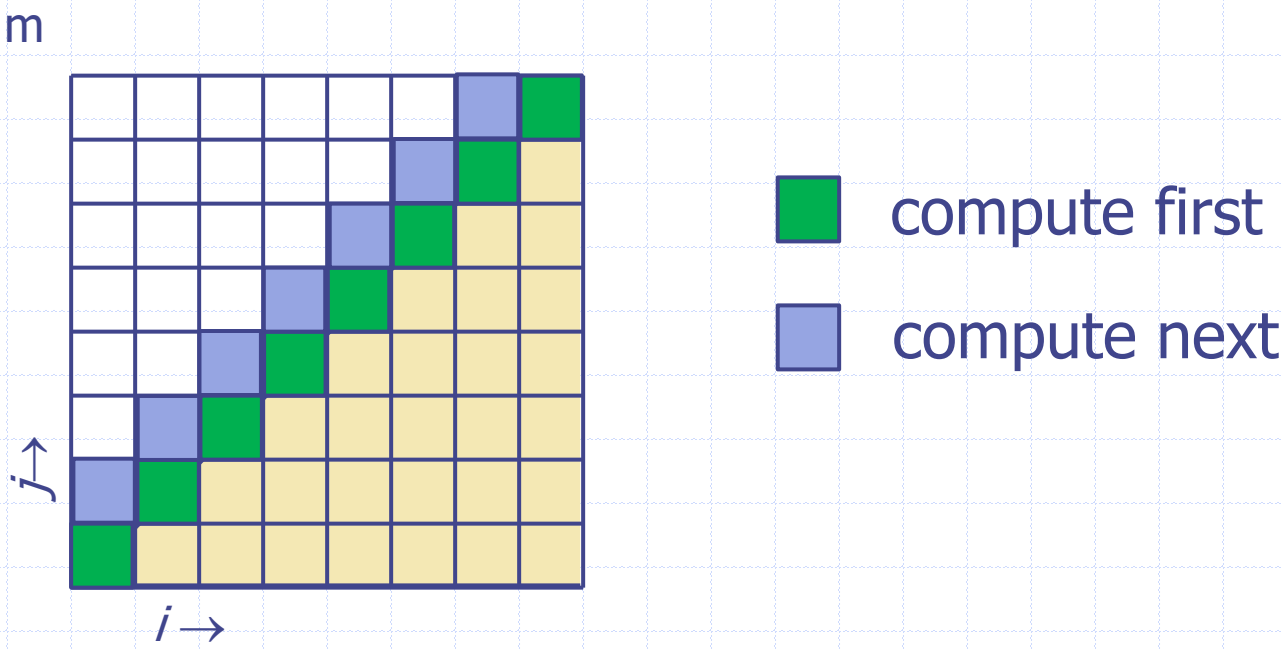
Dynamic Programming

In computing m[i, j] we use all values m[i, k] and m[k+1, j]. Therefore we must have those entries computed before computing m[i, j]. Visually:



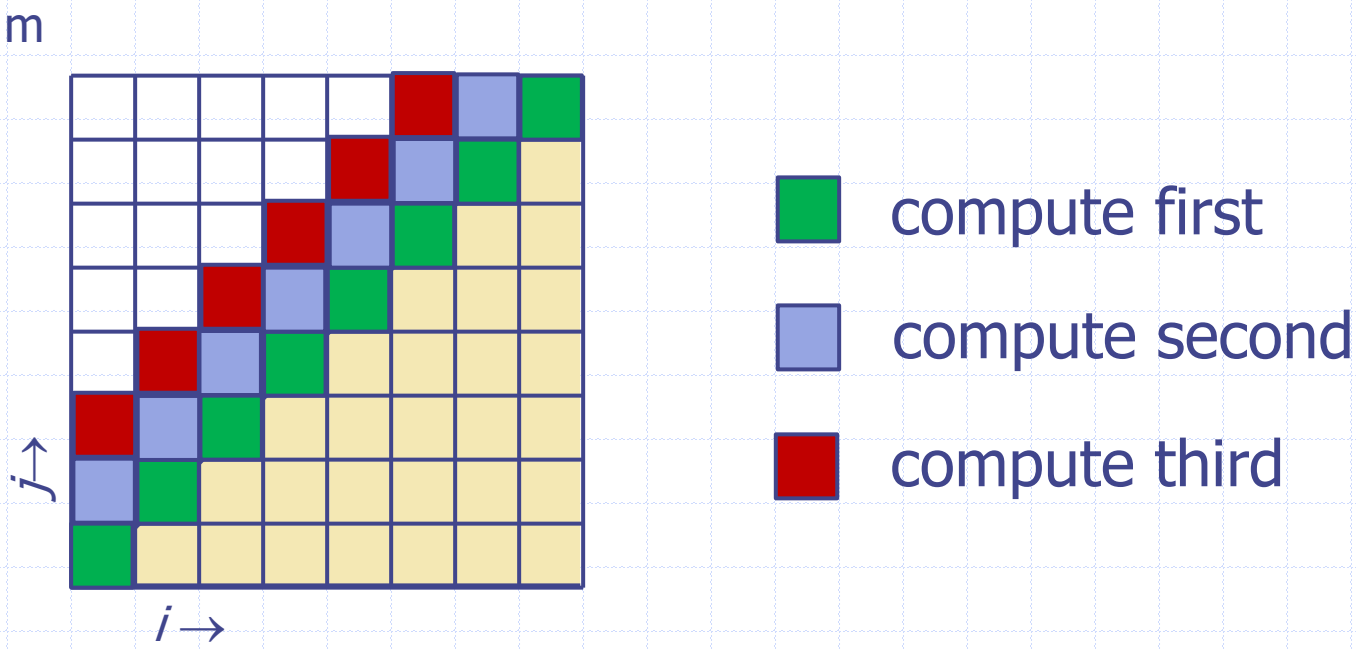
Dynamic Programming

The entries we can compute after we compute the i=j entries are the i=j-1 entries. These entries only involve subproblems with i=j.



Dynamic Programming

Then we can compute the entries with the i=j-2. These entries only involve subproblems with i=j or i=j-1.



Dynamic Programming

This can continue up until we have i = j – (n-1).

Define the length of a subproblem to be j – i + 1. Thus a chain of one matrix (i=j) has length 1, a chain of two matrices (i=j+1) has length 2, etc.

We can compute the costs for the subproblems in order of increasing length.

Dynamic Programming

```
solution() {
    allocate matrix m[1..n, 1..n]
    for i = 1 to n
        m[i, i] = 0;            // length 1

    for length = 2 to n {
        for i = 1 to n – length + 1 {
            j = i + length – 1
            m[i, j] = c(i, j);
        }
    }
    return c(1, n);
}
```

```
c(i, j) {
    min = ∞;
    for k = i to j-1 {
        cost = m[i, k] + m[k+1, j] +
                    p[i-1]*p[k]*p[j];
        if (cost < min)
            min = cost;
    }
    return min;
}
```

Compare this with *Matrix-Chain-Order* in text.

Dynamic Programming Traceback

Traceback can again be done the same way for DP as was done for memoization.

The text uses the method of storing choices, in an array s. This is included in *Matrix-Chain-Order* and in *Print-Optimal-Parens*.

Note that dynamic programming requires you to figure out the order in which to compute the table entries, but memoization does not.

(Memoization is itself straightforward enough that there are some languages or packages that will do it for you automatically.)