Dynamic Programming

Algorithm design technique

(The word 'programming' in title has nothing to do with computer programming)

Example

A_1 A_2

$m \times n$ $n \times p$

Total no. of multiplications

$m \cdot n \cdot p$

A_1 A_2 A_3 A_4 $\quad 2 \times 2$

2×3 3×5 5×10 10×2

$(A_1 \cdot A_2) \cdot (A_3 \cdot A_4) = 2 \times 3 \times 5 + 5 \times 10 \times 2 + 2 \times 5 \times 2$

$30 + 100 + 20 = 150$

$A_1 \cdot ((A_2 \cdot A_3) \cdot A_4) = 3 \times 5 \times 10 + 3 \times 10 \times 2 + 2 \times 3 \times 2$

$150 + 60 + 12$

$= 222$

Product operation on matrices is associative, so there are many ways to bracket this chain (corresponding to different evaluation order), all of which will give the same answer.

Order of multiplications changes the no. of multiplications (on elements) required, but result is the same.

Problem Matrix chain multiplication

$$A_1 A_2 \cdots A_n$$

A_i has dim $p_i \times p_{i+1}$

(Final product has dimension $p_1 \times p_{n+1}$)

Find a parenthesization of this chain which leads to minimum no. of multiplication.

Naive Brute Force idea: Consider all bracketings and find minimum

no. of such bracketings is exponential c^n (for $c > 1$)

Structure in the problem

$$(A_1 \cdots A_k)(A_{k+1} \cdots A_n)$$

$1 \le k \le n-1$

$$m_{ij} = \min_{i \le l \le j-1} \{ m_{il} + m_{l+1,j} + p_i \times p_{l+1} p_{j+1} \}$$

m_{ij} is the min no of multiplication in $A_i \cdots A_j$

Problem of computing m_{ij} is expressed in terms of smaller subproblems.

```
mm-rec(P, i, j)        // Array P contains information about dimensions of A_l's
                       // A_l has dim p_l x p_{l+1}.   P[1,...,n]
                       // m_ij to be computed
  if (i == j)
      return 0
  temp = ∞
  for l = i to j-1 do
      a = mm-rec(P, i, l)
      b = mm-rec(P, l+1, j)
      c = a + b + P[i] · P[l+1] · P[j]
      if c < temp
          temp = c
  return temp
```

Time complexity Analysis

$T(m)$ is the time taken to find the minimum no. of multiplications for chain of length m

Length of the input chain: $j - i + 1$

$$T(1) \geq 1$$

$$T(m) \geq (T(1) + T(m-1) + 1) + (T(2) + T(m-2) + 1) + \vdots + (T(m-1) + T(1) + 1)$$

$$= 2\sum_{k=1}^{m-1} T(k) + (m-1)$$

$T(1) \geq 1$, $T(n) \geq 2 \sum_{k=1}^{n-1} T(k) + n$ (for $n > 1$)

Claim $T(n) \geq 2^{n-1}$ for all n.

Proof is by induction on n.

Base case $T(1) \geq 1 = 2^{1-1} = 2^0 = 1$

Induction step: $T(n) \geq 2 \sum_{k=1}^{n-1} 2^{k-1} + n$

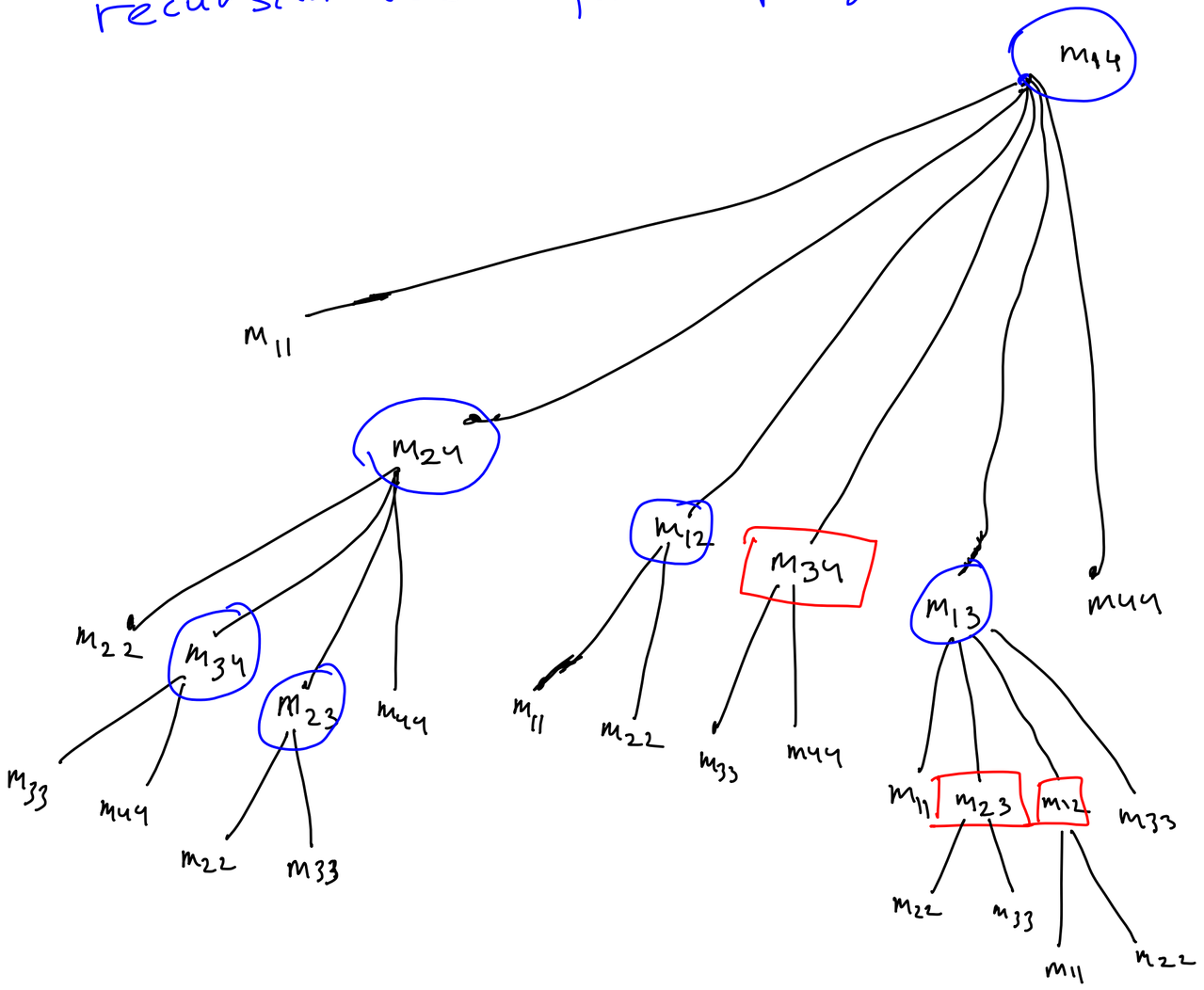
$= 2 + 2^2 + \cdots + 2^{n-1} + n$

$\geq 1 + 2 + 2^2 + \cdots + 2^{n-1} = 2^n - 1 \geq 2^{n-1}$

□

$T(n) \geq \frac{1}{2} \cdot 2^{(\text{size of the chain})}$ Exponential time

Let us analyse the cause of this exponential time recursion tree for computing m_{14}



Blue circles show calls made for the first time.

Red squares show calls which have been made before.

Our algorithm computes each call (Blue or red) from scratch.

Many subproblems are being computed several times.

Problem $A_1 \ldots A_n$ (compute m_{1n})

Subproblems: $A_i \ldots A_j$ (m_{ij})

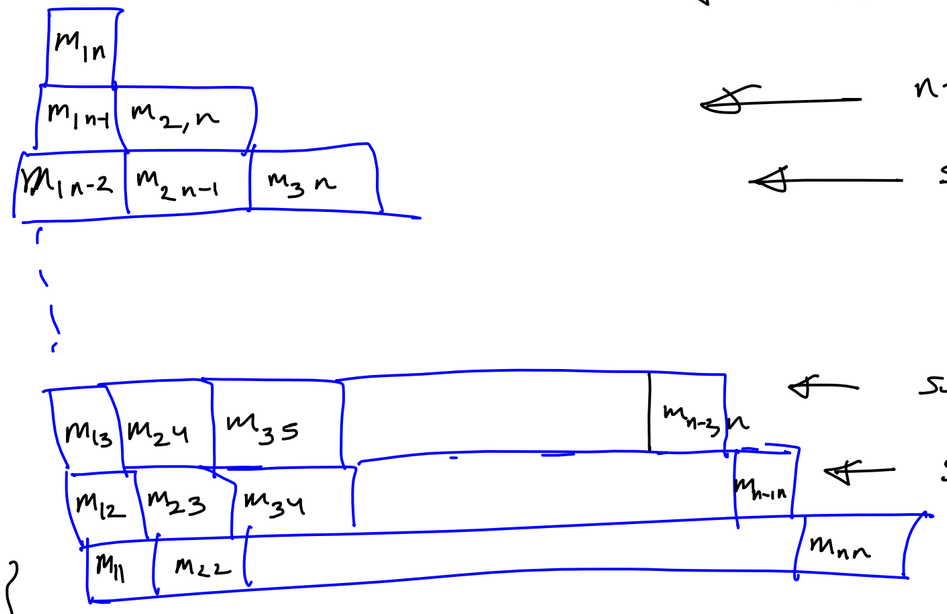
Total no. of subproblems in $O(n^2)$

Total no. of subproblems is small $O(n^2)$,

but recursion tree is exponential because, subproblems occur several times,

We can make the procedure more efficient by computing each subproblem only once and remembering its result.

The table is filled bottom up
(Row corresponding to length 1 is filled first then row corresponding to length 2
- - -
Row for n in the end)



← n

← n-1

← subproblems of length n-2

← subproblems of length 3

← subproblems of length 2

· subproblems of length 1

By our eqn

$$m_{ij} = \min_{1 \le k \le j-1} \left\{ m_{ik},\ m_{k+1\,j} + p_i p_{k+1} p_{j+1} \right\}$$

we can do this.
Expression for a row may be computed from entries in rows strictly below.

leads to $O(n^3)$ algorithm ⟶ polynomial time

['programming' in the title refers to tabular method.
For those who have heard of 'linear programming',
'programming' there also refers to tabular method]

Pseudo-code

```
matrix-chain-order-dp(P)
    n = P.length - 1
    m[1...n, 1...n], b[1...n, 1...n] are two new arrays
    for i = 1 to n do
        for j = 1 to n do
            m[i, j] = ∞

    for i = 1 to n do
        m[i, i] = 0

    for l = 1 to n-1 do // chains of length l+1
        for i = 1 to n-l do // chain A_{i, i+l}
            for j = i to i+l-1 do
                c = m[i, j] + m[j+1, i+l] + p_i · p_{j+1} · p_{i+l+1}
                if (m[i, i+l] > c)
                    m[i, i+l] = c
                    b[i, i+l] = j

    return m, b
```

At most three nested loops, each of which loops at most n times

$O(n^3)$ time complexity

Pseudo code for printing the optimal bracketing

```
opt-bracket(b, i, j)        // 1 ≤ i ≤ j ≤ n
                            // prints optimal bracketing of chain Ai ... Aj
  if (i == j)
      print (Ai)
      return
  l = b[i, j]
  if (i == l)
      print(Ai)
  else print("(")
       opt-bracket(b, i, l)
       print (")")
  if (l+1 == j)
       print (Aj)
  else print ("(")
       opt-bracket (b, l+1, j)
       print (")")
```