Dynamic Programming III [MIT OpenCourseware 6.006]

- Today's class:
  - Subproblems for strings
  - Parenthesization
  - Edit distance
  - Knapsack (how to pack your bags)
  - Pseudopolynomial time

- 5 "easy" steps for DP

① define subproblems
   #subproblems ≟ ? (better be polynomial)

② guess (part of solution)
   #guesses ≟ ? (better be polynomial)

③ Recurrence
   time/subproblem ≟ ?

④ Recurse + memoize
   (check that the recurrence is acyclic otherwise the computation will not terminate!)
   Total Time = #subproblems × time/subproblem

⑤ Solve original problem
   Check to see if the algorithm solves the original problem

Question:

The most difficult part with DP is defining the subproblems.

Is there anything that we can do to help us with this task?

Subproblems for strings/sequences

Remember:

- Text justification: sequence of words
- Blackjack: sequence of words

→ - Both cases used suffixes $X[i:]$ $\forall i$ ⇒ $n-i$ subproblems

$\Theta(n)$

- But sometimes prefixes $X[:i]$ $\forall i$ will need to be used ⇒ $n-i$ subproblems = $\Theta(n)$
- Other times: substrings $X[i:j]$ $\forall i \leq j$ $i,j \in [0, n]$

Q: Remember substrings?

A substring s' of a string s is an ordered sequence of consecutive elements of s

E.g.: "banana"

"ana" is a substring that appears twice in "banana"

Q: How many substrings do we have? Any ideas?

How many possible positions for i in the worst-case?

$i \leq n$

How many possible position for j in the worst-case?

- For each index i that is chosen there can be at most n possible values for index j (at most)
- There are n possible values for index i (at most)
- Therefore we have $n \times n = \Theta(n^2)$ possible combinations

---

Parenthesization Problem

- What is the optimal evaluation of associative expression.
- E.g. Matrix Multiplication

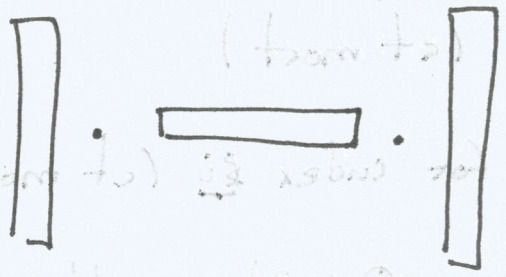
$$A_0 \cdot A_1 \cdot A_2 \cdots A_{n-1}$$

  - Not commutative
  - All that can be done is choose which multiplications (parenthesization) is performed first, e.g.

$$\left((A_0 \cdot A_1) \cdot (A_2 \cdots) A_{n-1} \right)$$

  - I want to try to minimize the total number of operations that are performed for matrix multiplication
  - Some parenthesizations will be cheaper than others

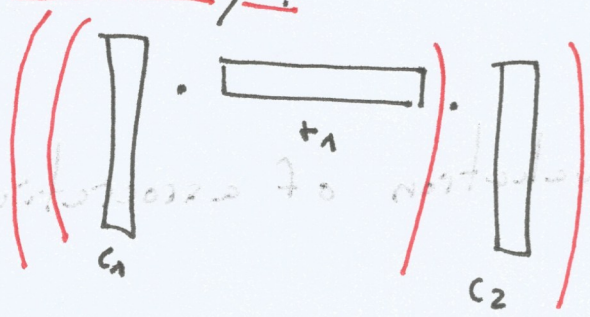
- Example:



(i.e. column vector * times a row vector * times a column vector)

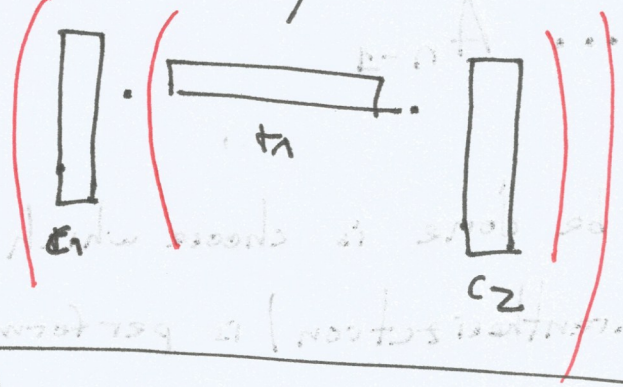
- There are two ways to compute this product:

- One way:



I.e.
- First multiply $c_1 \cdot r_1$
- Then multiply the result $(c_1 r_1)$ with c_2

- Another way:

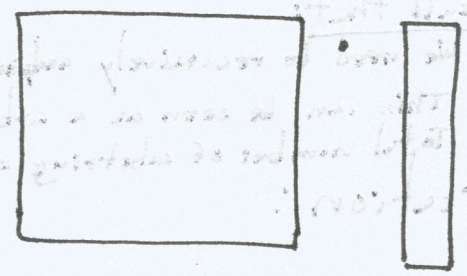


I.e.
- First multiply $r_1 \cdot c_2$
- Then multiply the result $(r_1 \cdot c_2)$ with c_1

Q: What is the difference? Any ideas?

- Assume each vector has n elements
  - $c_1 \cdot r_1$ = matrix of dimension n^2 $(n \times n)$
  - $(c_1 \cdot r_1) \cdot c_2$ = matrix of dimension n $(n \times 1)$
  - $r_1 \cdot c_2$ = matrix of dimension 1
  - $c_1 \cdot (r_1 \cdot c_2)$ = matrix of dimension n $(n \times 1)$

- The thing is:



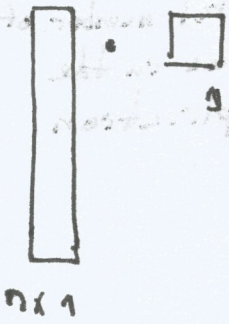
→ For each row of the $n \times n$ matrix we will need to perform:
- n multiplications
- $n-1$ additions

∴ 1 For all rows of the $n \times n$ matrix:

$$n \times (n + n - 1) = n^2 + n^2 - n = 2n^2 - n$$

(all rows; multiplications; additions)

$$\Theta(n^2) \text{ operations}$$

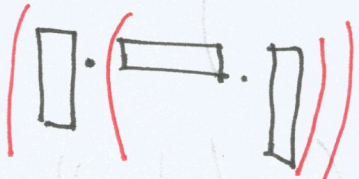


→ For each row of the $n \times 1$ column vector we need to perform 1 multiplication

∴ 2 For all rows of the $n \times 1$ column vector:

$$n \times 1 = n = \Theta(n) \text{ operations}$$

- Better if we use the parenthesization:



- Go to page 5.1 and 5.2

Question:

How can we solve this problem by dynamic programming?

- For example if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$ 5.1
what are the possible parenthesization scenarios:

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

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

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

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

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

- **Question:**

How many possible configurations would we have to check?

- Let $P(n)$ denote the number of alternative parenthesization of a sequence of n matrices.
- When $n = 1$ we have just one matrix and therefore only one way to parenthesize the result
- When $n \geq 2$ a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts

- The split may occur between the k^{th} and the $(k+1)^{st}$ matrices for any $k = 1, 2, \ldots, n-1$

- Thus we obtain the recurrence

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

- This recurrence is very similar to a sequence called the Catalan numbers which grows as $\Omega\left(\frac{4^n}{n^{3/2}}\right)$

The exponential grow is going to dominate over $n^{1.5}$ ...

Q: How can we solve this problem by dynamic programming?

① Subproblem = optimal evaluation of $A_i \dots A_{j-1}$

#subproblems = $\Theta(n^2)$

Recall that:
- We need to recursively analyze $A_i \dots A_{j-1}$
- This can be seen as a substring
- Total number of substrings = $\Theta(n^2)$

② Guess: outermost / last multiplication:

$$(A_i \dots A_{k-1}) \cdot (A_k \dots A_{j-1})$$

- We are going to guess what will be the index k
- # choices for k = $O(j-i+1) = O(n)$

Recall that n is the total number of matrices in the multiplication

③ Recurrence:

$$DP(i,j) = \min\Big(DP(i,k-1) + DP(k,j-1) + \text{"cost of } (A_i : A_{k-1}) \cdot (A_k : A_{j-1})\text{"} \text{ for } k \text{ in range } (i+1, j) \Big)$$

We just need to develop a simple function to count the number of operations. Assume this can be done in $\Theta(1)$ time

- time / subproblem = $O(n)$

Recall that:
- Recursive calls will be "free", i.e. $\Theta(1)$
- Major cost will be from the guessing the k cycle where we need to try $O(n)$ guesses

④ time = #subproblems × time / subproblem

$$= \Theta(n^2) \times O(n) = O(n^3)$$

With further analysis we can show that this would actually be $\Theta(n^3)$