Analysis of Algorithms
Part II: Programming paradigms

Programming paradigms

Brute force approach

Divide and conquer

Greedy algorithms:
activity selection problem
fractional knapsack problem

Dynamic programming:
matrix chain multiplication problem
longest common subsequence problem

Programming paradigms

- *programming paradigm* = general approach to algorithm design

  = general philosophy for solving a problem

there are different ways of solving a problem

→ several possible programming paradigms to solve the same problem

a programming paradigm may be adapted to a certain problem and not to another

Brute force approach

- *most direct approach to solving a problem*

  → we do not try to minimize the running time

searching problem = within a (big) set of possible solutions, search for the "best" solution

→ brute force approach consists in testing systematically all the possible solutions

- *traveling salesman problem*

first call: ***all_possible_paths(list of all cities , empty path)***

```
all_possible_paths( list_cities , path )
   if list_cities is empty
      calculate total distance of path
      and check if it is the shortest path so far
   else
      for each city inside list_cities
         all_possible_paths( list_cities without city , path with city added to it )
```

example:

list of all cities: (Istanbul , Mersin , Bursa)

n = 3 → 3! = 3 × 2 × 1 = 6 possible paths

path 1:	Istanbul	Mersin	Bursa
path 2:	Istanbul	Bursa	Mersin
path 3:	Mersin	Istanbul	Bursa
path 4:	Mersin	Bursa	Istanbul
path 5:	Bursa	Istanbul	Mersin
path 6:	Bursa	Mersin	Istanbul

Divide and conquer

- *step 1:* *divide* the problem into *subproblems*
- *step 2:* *solve* each subproblem *independently*
- *step 3:* *combine* the subproblems' solutions to obtain the problem's solution

apply the same procedure to solve each subproblem

→ *recursive algorithm*

binary search:

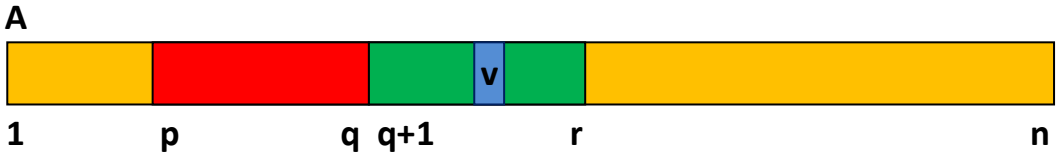
search problem: find value **v** inside **A[1..n]**

→ *method:*

- *step 1:* sort **A** in increasing order
- *step 2:* binary search within **A**

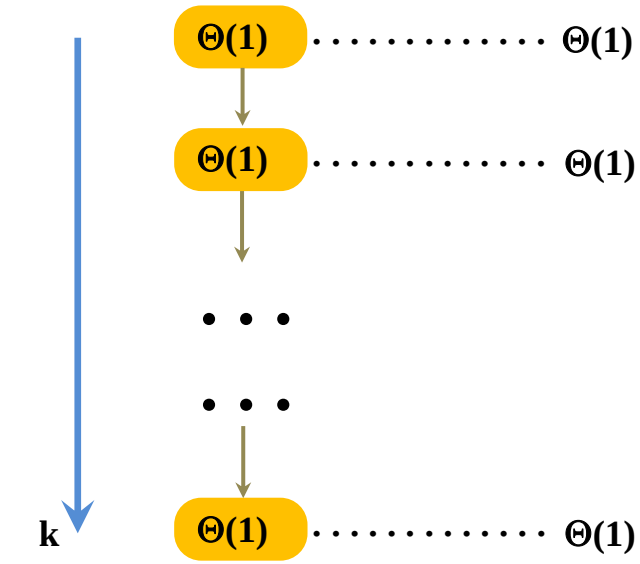
first call: **binary_search(A , 1 , n , v)**

```
binary_search(A , p , r , v)
    if p = r
        if v = A[p]  return p
                else  return NIL
    else
        q ← ⌊ (p + r) / 2 ⌋
        if v ≤ A[q]  return binary_search(A , p      , q , v)
                else  return binary_search(A , q + 1 , r , v)
```



- *analysis and comparison with linear search:*

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n/2) + \Theta(1) & \text{if } n > 1 \end{cases}$$



$n / 2^k \approx 1$

$\Rightarrow k \approx \lg n$

$\Rightarrow T(n) = k \times \Theta(1) = \lg n \times \Theta(1)$

$\Rightarrow T(n) = \Theta(\lg n)$

- *first*, preliminary sorting with *merge sort* in $\Theta(n \lg n)$
- *then*, *binary search* in $\Theta(\lg n)$

linear search in $\Theta(n)$

assume data is sorted *once and for all*, then *many* binary searches are performed over it

→ binary search becomes much more interesting than linear search

merge sort:

sorting problem: sort **A[1..n]** in ascending order

- divide the sequence of values into two subsequences
- sort each subsequence independently
- merge the two sorted subsequences into the sorted sequence

first call: **merge_sort(A , 1 , n)**

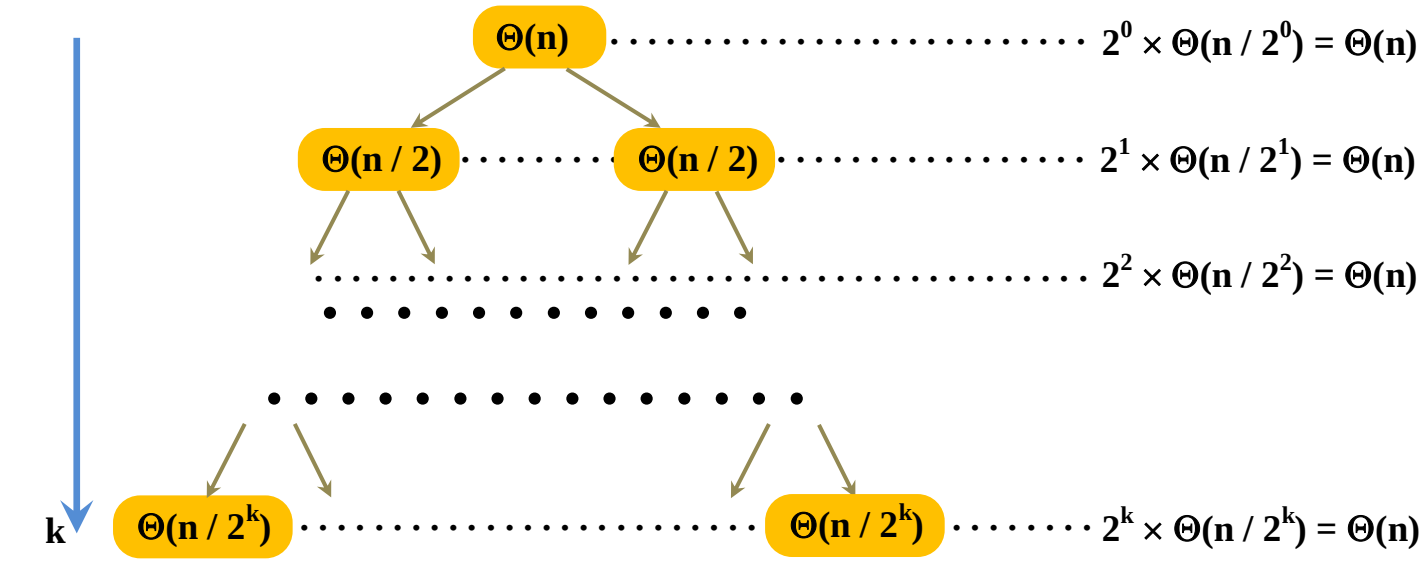
```
merge_sort(A , p , r)
    if p < r
        q ← ⌊ (p + r) / 2 ⌋
        merge_sort(A , p , q)
        merge_sort(A , q + 1 , r)
        merge(A , p , q , r)
```

Θ(n)

$$\Rightarrow \quad T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2\,T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

(assuming **n = r - p + 1**)

- *analysis and comparison with insertion sort:*



$n / 2^k \approx 1$

$\Rightarrow k \approx \lg n$

$\Rightarrow T(n) = k \times \Theta(n) = \lg n \times \Theta(n)$

$\Rightarrow T(n) = \Theta(n \lg n)$

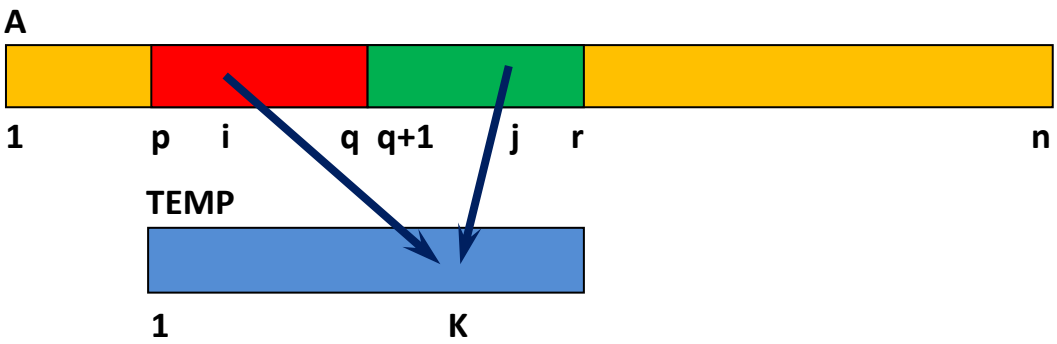
→ much better than *insertion sort*
which runs in $O(n^2)$ ($\Theta(n^2)$ in worst case)

```
merge(A , p , q , r)
    i ← p  ;  j ← q + 1  ;  K ← 1   // merge the intertwined elements of A[p..q] , A[q+1..r]
    while  i ≤ q  and  j ≤ r
        if A[i] ≤ A[j]
            TEMP[K] ← A[i]   ;   i ← i + 1
        else
            TEMP[K] ← A[j]   ;   j ← j + 1
        K ← K + 1

    while i ≤ q        // copy remaining elements of A[p..q]
        TEMP[K] ← A[i]   ;   i ← i + 1   ;   K ← K + 1

    while j ≤ r        // copy remaining elements of A[q+1..r]
        TEMP[K] ← A[j]   ;   j ← j + 1   ;   K ← K + 1

    for k ← 1 to K - 1
        A[p - 1 + k] ← TEMP[k]
```



Greedy algorithms

optimization problem = *searching problem*: we must find an optimal solution among all possible solutions

optimal substructure property: an optimal solution to the problem is made up of optimal solutions to subproblems

→ algorithm for optimization problem = sequence of steps at which *choices* must be made

- at each step, a *greedy algorithm* makes the choice that seems the best at the moment

  = *locally optimal choice*

  = *narrow-minded choice*

- for certain categories of problems(*activity selection*, *fractional knapsack*, ...), such a greedy approach *always* provides an optimal solution

  = not the case for other categories of problems... (such as *matrix chain multiplication*)

activity selection problem:

schedule a resource among several competing activities

n activities (*lectures*...) that wish to use a single resource (*lecture room*...)

each activity **i** $(1 \le i \le n)$ { starts at time s_i ; finishes at time f_i

→ activity **i** takes place during time interval $[s_i , f_i)$

activities **i** and **j** do not *clash* if either $s_i \ge f_j$ or $s_j \ge f_i$ (one starts after the other has finished)

problem: select a maximum of activities that do *not* clash

a greedy approach *always* provides an optimal solution to the activity selection problem

- *greedy procedure:*

step 1: sort the activities in order of increasing finish time $(f_1 \le f_2 \le \ldots \le f_n)$

→ the first activities in the list are those that finish first

step 2: run the greedy code below:

set of selected activities

latest selected activity

```
greedy_activity_selector( s , f , n )
    A ← { 1 }
    j ← 1
    for i ← 2 to n
        if  s[i] ≥ f[j]
            A ← A ∪ { i }
            j ← i
    return A
```

select, out of the remaining activities, the one that ***finishes first*** (scan list in order) and that *does not clash* with the already selected activities (enough to check **j** only)
→ leaves a maximum of remaining time to schedule extra activities

analysis:

- sorting in **Θ(n lg n)**
- greedy code in **Θ(n)**

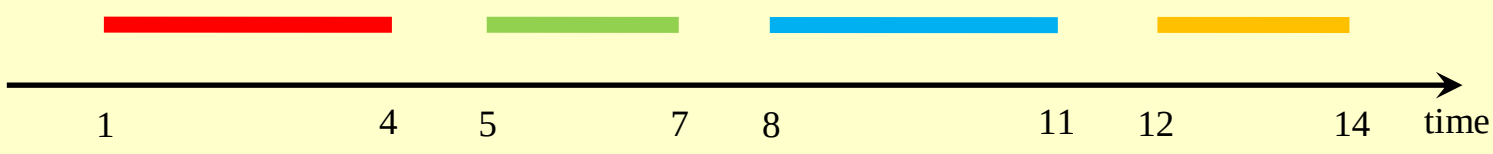
→ greedy approach is very efficient

example:

start time	12	1	3	2	8	8	6	5	3	5	0
finish time	14	4	5	13	12	11	10	9	8	7	6

→ sort in order of increasing finish time:

i	1	2	3	4	5	6	7	8	9	10	11
s[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14



fractional knapsack problem:

a thief robbing a store finds **n** items

each item **i** (**1≤ i ≤ n**) { is worth v_i euros (€)
weighs w_i kilograms

the robber can carry away at most **W** kilograms in his knapsack

if necessary, the robber can take away only a *fraction* of an item

problem: select a list of items (eventually a fraction) whose *total weight* is no more than **W**

and whose *total value* is maximal

a greedy approach *always* provides an optimal solution to the fractional knapsack problem

- *greedy procedure:*

```
greedy_fractional_knapsack( v , w )
    for each item i
        compute  v_i / w_i    (value per kilogram)

    while there are some items remaining
            and some knapsack capacity left
        take as much as possible of the item
        with the greatest value per kilogram
```

- *analysis:*

selection of items in **Θ(n)**

→ greedy approach is very efficient

example:

knapsack of capacity **W** = 50 kg

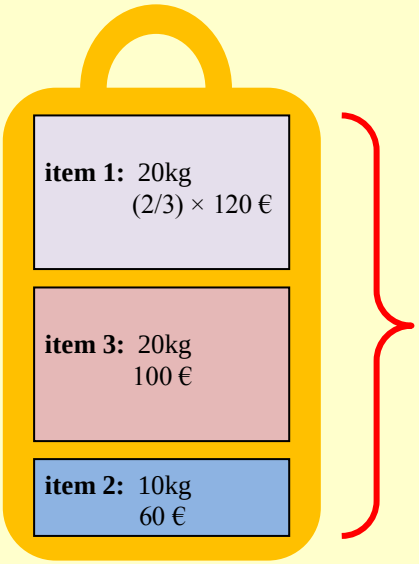
item	1	2	3
value (€)	120	60	100
weight (kg)	30	10	20
value per kg	4	6	5

step 1: take all of **item 2** → 50 - 10 = 40 kg of capacity left

step 2: take all of **item 3** → 40 - 20 = 20 kg of capacity left

step 3: take 20 kg of **item 1** and leave behind 10 kg of it

→ resulting knapsack:



total value = 60 € + 100 € + 80 €
= 240 €
= highest possible value

Dynamic programming

divide and conquer: { subdivide recursively the problem into subproblems
solve these subproblems *independently* from each other

with certain categories of problems, some of these subproblems *occur several times* during the recursive subdivision

→ since the subproblems are solved independently,
we keep solving again and again the same subproblems without knowing it !

- *dynamic programming:*

similarly to *divide and conquer*, we subdivide recursively into subproblems

but these subproblems are no longer solved independently!

the *first time* a subproblem is encountered, it is solved *once and for all*
and its solution is stored in a *data structure* (*table*)

→ the other times the same subproblem is encountered,
we simply read its solution in the data structure

- *in general, dynamic programming is applied to optimization problems:*

a *value* is associated to each of the possible solutions

→ we must find a solution whose value is *optimal* (minimal or maximal)

- *an optimization problem must have two properties if we want to solve it with dynamic programming:*

  optimal substructure property: an optimal solution to the problem must be made up of optimal solutions to subproblems

  → recursive resolution of subproblems that are themselves optimization problems

  relatively few subproblems that occur many times

  → algorithmic complexity substantially reduced

- *procedure to develop a dynamic programming algorithm:*

  step 1: define recursively an optimal solution in function of optimal solutions to subproblems

  step 2: define recursively the value of an optimal solution (recurrent scheme)

  step 3: compute the value of an optimal solution with an iterative algorithm

  step 4: construct an optimal solution

matrix chain multiplication problem:

- *presentation of the problem:*

we are given a sequence (chain) of **n** *rectangular* matrices $\mathbf{A_1}, \mathbf{A_2}, \ldots, \mathbf{A_n}$
and we have to calculate the product $\mathbf{A_1\, A_2 \ldots A_n}$

matrix multiplication is *associative*: $\mathbf{A\, B\, C = (\,(\,A\, B\,)\, C\,) = (\,A\,(\,B\, C\,)\,)}$

example:

$$\mathbf{A_1\, A_2\, A_3\, A_4} = \mathbf{(\,A_1\,(\,A_2\,(\,A_3\, A_4\,)\,)\,)} = \mathbf{(\,(\,(\,A_1\, A_2\,)\, A_3\,)\, A_4\,)}$$

$$= \mathbf{(\,A_1\,(\,(\,A_2\, A_3\,)\, A_4\,)\,)} = \mathbf{(\,(\,A_1\,(\,A_2\, A_3\,)\,)\, A_4\,)}$$

$$= \mathbf{(\,(\,A_1\, A_2\,)\,(\,A_3\, A_4\,)\,)}$$

→ many possible parenthesizations for product $\mathbf{A_1\, A_2 \ldots A_n}$

→ in which order will we chose to multiply two by two all these matrices ?

the choice of a parenthesization may have a huge effect
on the total *number of floating point multiplications* necessary to calculate $\mathbf{A_1\, A_2 \ldots A_n}$

note: **A (p × q)** and **B (q × r)** ⇒ **A B** requires **p q r** floating point multiplications

example:

A (10 × 100) , **B (100 × 5)** , **C (5 × 50)**

((A B) C) requires $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ floating point multiplications

(A (B C)) requires $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$ floating point multiplications

→ we must find a parenthesization for $\mathbf{A_1\, A_2 \ldots A_n}$, among all possible ones,
that minimizes the total *number of floating point multiplications*

= *optimal parenthesization*

there may be several parenthesizations that share the *same optimal cost*

- *what if we used brute force ?. . . :*

P(n) = total number of possible parenthesizations for a product of **n** matrices $\mathbf{A_1\, A_2 \ldots A_n}$

parenthesization = *recursive subdivision* of $\mathbf{A_1\, A_2 \ldots A_n}$ into subproducts

n - 1 possible ways of splitting $\mathbf{A_1\, A_2 \ldots A_n}$ into two subproducts:

$$\mathbf{A_1\, A_2 \ldots A_n} \rightarrow \begin{cases} \mathbf{A_1 \ldots A_k} \\ \mathbf{A_{k+1} \ldots A_n} \end{cases} \qquad (\mathbf{k = 1, 2, \ldots, n - 1})$$

each of these two subproducts is itself parenthesized *independently* from the other, hence:

$$\mathbf{P(n)} = \begin{cases} \mathbf{1} \quad \textit{if} \ \mathbf{n = 2} \\ \mathbf{P(1) \times P(n - 1) + P(2) \times P(n - 2) + \ldots + P(n - 1) \times P(1)} \quad \textit{if} \ \mathbf{n > 2} \end{cases}$$

$$\Rightarrow \quad \mathbf{P(n) = \Omega(4^n / n^{3/2})}$$

P(n) grows *exponentially* with **n**

brute force = test each of the possible solutions = *intractable task !*

- *application of dynamic programming to the matrix chain multiplication problem:*

step 1:

recursively, we must parenthesize $\mathbf{A_i \ldots A_j}$

→ we split it in two:

$$\mathbf{A_i \ldots A_j} = \mathbf{(A_i \ldots A_k)} \times \mathbf{(A_{k+1} \ldots A_j)} \quad \textit{with} \quad \mathbf{k} = i, \ldots, \mathbf{j - 1}$$

optimal substructure property:

to parenthesize optimally $\mathbf{A_i \ldots A_j}$, we must also parenthesize optimally $\mathbf{A_i \ldots A_k}$ and $\mathbf{A_{k+1} \ldots A_j}$

step 2:

notations:

- list of matrix sizes: $\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_n}$ → matrix $\mathbf{A_i}$ has size $\mathbf{p_{i-1} \times p_i}$
- the resulting matrix of product $\mathbf{A_i \ldots A_j}$ is noted $\mathbf{A_{i..j}}$
- the optimal cost of calculating $\mathbf{A_{i..j}}$ is noted $\mathbf{m_{ij}}$ = minimal number of flot. point. mult.

optimal cost for $\mathbf{A_{i..j}}$ = *optimal cost* for $\mathbf{A_{i..k}}$

\+ *optimal cost* for $\mathbf{A_{k+1..j}}$

\+ *cost* of multiplying $\mathbf{A_{i..k}}$ $(\mathbf{p_{i-1} \times p_k})$ with $\mathbf{A_{k+1..j}}$ $(\mathbf{p_k \times p_j})$

. . . . . *assuming that* we have chosen the value for **k** that corresponds to the optimal cost. . .

→ recurrent scheme: (*you must memorize it*)

$$\mathbf{m_{ij}} = \begin{cases} \mathbf{0} & \text{if } \mathbf{i = j} \\ \min\limits_{\mathbf{i \le k < j}} \{ \mathbf{m_{ik} + m_{k+1,j} + p_{i-1} \times p_k \times p_j} \} & \text{if } \mathbf{i < j} \end{cases}$$

step 3:

establish an *iterative* algorithm that implements the recurrent scheme of $\mathbf{m}_{i\,j}$

→ fill up tables
- **m[i , j]** = optimal cost of calculating $\mathbf{A}_{i\,..\,j}$
- **s[i , j]** = value of **k** for optimal cost = *best splitting place*

dependencies between cases of **m** → fill tables **m** and **s** by *order of increasing chain length* **L**

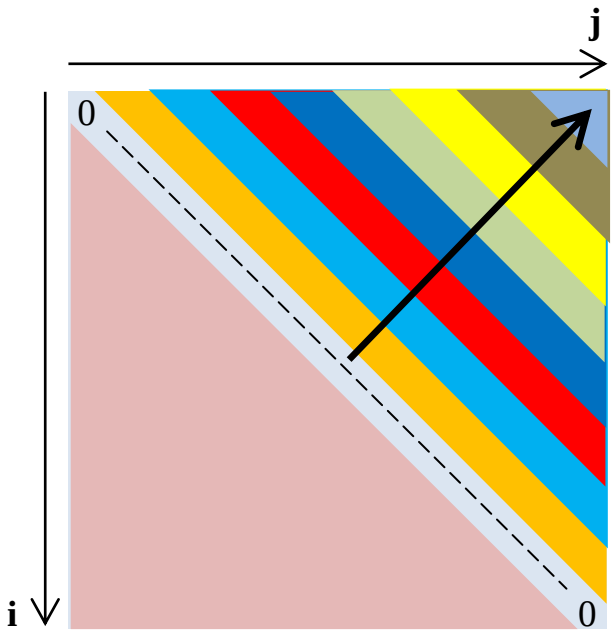
- *step 1:* fill the main diagonal of **m** with 0 values
- *step 2:* fill the first upper diagonal of **m**
- *step 3:* fill the second upper diagonal of **m**
- . . . . . .
- *step n:* fill the case in the upper right corner of **m**

```
matrix_chain_order(p , n)
    for i ← 1 to n
        m[i , i] ← 0

    for L ← 2 to n
        for i ← 1 to n - L + 1
            j ← i + L - 1
            m[i , j] ← + ∞

            for k ← i to j - 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
                    s[i , j] ← k

    return m and s
```



analysis:

filling of tables **m** , **s** in $\Theta(n^3)$

memory required for **m** , **s** in $\Theta(n^2)$

step 4:

calculate $\mathbf{A_{1\,..\,n}}$ with the optimal parenthesization described in table **s**

→ read table **s** to calculate recursively $\mathbf{A_{i\,..\,j}}$ as $\mathbf{A_{i\,..\,s[i,j]} \times A_{s[i,j]+1\,..\,j}}$

first call: **matrix_chain_multiply(A , s , 1 , n)**

```
matrix_chain_multiply( A , s , i , j )
    if i = j
        return A[i]
    else
        X ← matrix_chain_multiply( A , s , i , s[i , j] )
        Y ← matrix_chain_multiply( A , s , s[i , j] + 1 , j )
        return mul_mat( X , Y , p[i - 1] , p[s[i , j]] , p[j] )
```

example:

matrix	A_1	A_2	A_3	A_4
size	2 × 2	2 × 3	3 × 4	4 × 2

i	0	1	2	3	4
p_i	2	2	3	4	2

$m_{13} = \min \{ m_{11} + m_{23} + p_0 p_1 p_3 , m_{12} + m_{33} + p_0 p_2 p_3 \}$
$= \min \{ 0 + 24 + 2 \times 2 \times 4 , 12 + 0 + 2 \times 3 \times 4 \}$

→ table **m**:

	1	2	3	4
1	0	12	36	44
2		0	24	36
3			0	24
4				0

$m_{14} = \min \{ m_{11} + m_{24} + p_0 p_1 p_4 , m_{12} + m_{34} + p_0 p_2 p_4 , m_{13} + m_{44} + p_0 p_3 p_4 \}$
$= \min \{ 0 + 36 + 2 \times 2 \times 2 , 12 + 24 + 2 \times 3 \times 2 , 36 + 0 + 2 \times 4 \times 2 \}$

$m_{12} = m_{11} + m_{22} + p_0 p_1 p_2 = 0 + 0 + 2 \times 2 \times 3$

$m_{23} = m_{22} + m_{33} + p_1 p_2 p_3 = 0 + 0 + 2 \times 3 \times 4$

$m_{24} = \min \{ m_{22} + m_{34} + p_1 p_2 p_4 , m_{23} + m_{44} + p_1 p_3 p_4 \}$
$= \min \{ 0 + 24 + 2 \times 3 \times 2 , 24 + 0 + 2 \times 4 \times 2 \}$

$m_{34} = m_{33} + m_{44} + p_2 p_3 p_4 = 0 + 0 + 3 \times 4 \times 2$

→ table **s**:

	1	2	3	4
1		1	2	1
2			2	2
3				3
4				

→ optimal parenthesization: **(A_1 (A_2 (A_3 A_4)))** requires only **m[1 , 4]** = 44 f. p. mul.

∧ k=1 ∧ k=2 ∧ k=3

Longest Common Subsequence (*LCS*) problem:

- *presentation of the problem:*

subsequence of a given sequence (list) of elements:

> pick up some elements of the sequence and keep them in their original order

example:

sequence **< A , B , C , B , D , A , B >**

→ some subsequences: **< A , C , A , B >**
< B , C , B , D >
< B , C , A >
< B , C , B , A >
< C , D , A >
. . . . . .

common subsequence of two given sequences:

> subsequence of both sequences

example:

sequences **< A , B , C , B , D , A , B >** and **< B , D , C , A , B , A >**

→ some common subsequences: **< C , A , B >**
< A , B , A >
< B , C , A >
< B , C , B , A >
. . . . . .

Longest Common Subsequence problem:

> find a maximum length common subsequence of two given subsequences

- *what if we used brute force ?. . . :*

  subsequence of $\mathbf{X = < x_1 , x_2 , . . . , x_m >}$

  → each element $\mathbf{x_i}$ may be picked up or not (2 possibilities), independently from the others

  → there are $2 \times 2 \times . . . \times 2 = 2^m$ possible subsequences

  the number of subsequences grows *exponentially* with the size of the sequence

brute force = test each of the possible solutions = *intractable task !*

- *application of dynamic programming to the LCS problem:*

step 1:

Notation: *i-th prefix* of $\mathbf{X} = \langle \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_m} \rangle$ is $\mathbf{X_i} = \langle \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_i} \rangle$

Theorem: assume $\mathbf{X} = \langle \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_m} \rangle$ and $\mathbf{Y} = \langle \mathbf{y_1}, \mathbf{y_2}, \ldots, \mathbf{y_n} \rangle$

then, let $\mathbf{Z} = \langle \mathbf{z_1}, \mathbf{z_2}, \ldots, \mathbf{z_k} \rangle$ be any *LCS* of $\mathbf{X}$ and $\mathbf{Y}$

$\mathbf{Z}$ has the following properties:

- if $\mathbf{x_m} = \mathbf{y_n}$: $\mathbf{z_k} = \mathbf{x_m} = \mathbf{y_n}$ and $\mathbf{Z_{k-1}}$ is an *LCS* of $\mathbf{X_{m-1}}$ and $\mathbf{Y_{n-1}}$
- if $\mathbf{x_m} \neq \mathbf{y_n}$:
  - if $\mathbf{z_k} \neq \mathbf{x_m}$: $\mathbf{Z}$ is an *LCS* of $\mathbf{X_{m-1}}$ and $\mathbf{Y}$
  - if $\mathbf{z_k} \neq \mathbf{y_n}$: $\mathbf{Z}$ is an *LCS* of $\mathbf{X}$ and $\mathbf{Y_{n-1}}$

→ *optimal substructure property*:

an *LCS* of two sequences is made up of an *LCS* of prefixes of the two sequences

step 2:

in practice:
- if $\mathbf{x_m} = \mathbf{y_n}$, then find an an *LCS* of $\mathbf{X_{m-1}}$ and $\mathbf{Y_{n-1}}$
- if $\mathbf{x_m} \neq \mathbf{y_n}$, then find an *LCS* of $\mathbf{X_{m-1}}$ and $\mathbf{Y}$, find an *LCS* of $\mathbf{X}$ and $\mathbf{Y_{n-1}}$ and take the longest of the two

→ recurrent scheme: (*you must memorize it*)

let $\mathbf{c_{i\,j}}$ be the length of an *LCS* of the prefixes $\mathbf{X_i}$ and $\mathbf{Y_j}$ = optimal value

$$\mathbf{c_{i\,j}} = \begin{cases} \mathbf{0} & \text{if } \mathbf{i = 0} \text{ or } \mathbf{j = 0} \\ \mathbf{c_{i-1,\,j-1} + 1} & \text{if } \mathbf{i, j > 0} \text{ and } \mathbf{x_i = y_j} \\ \mathbf{\max(\, c_{i,\,j-1}, c_{i-1,\,j}\,)} & \text{if } \mathbf{i, j > 0} \text{ and } \mathbf{x_i \neq y_j} \end{cases}$$

step 3:

establish an *iterative* algorithm that implements the recurrent scheme of $\mathbf{c_{i\,j}}$

→ fill up tables **c[0..m , 0..n]** and **b[0..m , 0..n]** for **X** of size **m** and **Y** of size **n**

- **c[i , j]** = maximal length of a common subsequence of prefixes $\mathbf{X_i}$ and $\mathbf{Y_j}$
- **b[i , j]** = points to the case of the chosen subproblem ($\mathbf{c_{i-1\,,\,j-1}}$ or $\mathbf{c_{i\,,\,j-1}}$ or $\mathbf{c_{i-1\,,\,j}}$)

dependencies between cases of **c** → fill tables **c** and **b** in *row-major order*

```
LCS_length(X , m , Y , n)
    for i ← 0 to m
        c[i , 0] ← 0

    for j ← 1 to n
        c[0 , j] ← 0

    for i ← 1 to m
        for j ← 1 to n
            if X[i] = Y[j]
                c[i , j] ← c[i - 1 , j - 1] + 1
                b[i , j] ← '↖'
            else
                if c[i - 1 , j] ≥ c[i , j - 1]
                    c[i , j] ← c[i - 1 , j]
                    b[i , j] ← '↑'
                else
                    c[i , j] ← c[i , j - 1]
                    b[i , j] ← '←'
    return c and b
```

analysis:

filling of tables **c** , **b** in **Θ(m n)**

memory required for **c** , **b** in **Θ(m n)**

step 4:

- begin at **b[m , n]** and trace through the table following the arrows
- whenever a '↖' is encountered, add the corresponding element at the head of a list

for manual tracing of the algorithm, we will in practice use only one table that will contain both **c[i , j]** and the arrows

example: **X** = < A , B , C , B , D , A , B > and **Y** = < B , D , C , A , B , A >

		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

→ *LCS*(**X** , **Y**) = < B , C , B , A >