Design and Analysis of Algorithms

Dynamic Programming

- **Rod Cutting**
- **Weighted interval scheduling**
- **Longest Common Subsequence**
- **Chain matrix multiplication**

Zahed Rahmati
Dept of Math and CS
Tehran Polytechnic

Introduction

- Dynamic Programming (DP) is among the ***most powerful techniques*** for designing algorithms for **optimization problems**.

- **Like DC** method**:**
  - **Break problem** into **smaller subproblems**, which can be solved **recursively**.
  - Thus it **solves problems by combining the solutions to subproblems.**
- **Unlike DC** method,
  - In **Dynamic Programming (DP)** the **subproblems typically *overlap* each other**.
  - (In **DC** the **subproblems are *disjoint***)

Introduction

- "Programming" in this context refers to a ***tabular method***, not to writing computer code.
- A DP algorithm **solves each subsubproblem just once** and then **saves its answer in a table**, avoiding the work of recomputing the answer every time it solves each subsubproblem.

DP solutions rely on two important structural qualities:

- **Optimal substructure**:
  - **An optimal solution to a problem (instance) contains optimal solutions to subproblems;** Each subproblem should be solved optimally in order to solve the global problem optimally.
- **Overlapping Subproblems**:
  - **A recursive solution contains a small number (polynomial in *n*) of distinct subproblems repeated many times**.

Solving DP subproblems

How to generate the solutions to these subproblems?

- **Top-Down *with memoization***:
  - In this approach, we write **a procedure that applies recursion** to solve the problem.
    - The procedure **checks to see whether it has previously solved this subproblem**: **If so,** it returns the saved value, **If not**, the procedure computes the value in the usual manner.
  - **Memoization** **records** the results of recursive calls in a table so that subsequent calls to a previously solved subproblem check the table to avoid redoing work.
- **Bottom-up**:
  - The **solution is built iteratively** by combining the solutions to small subproblems to obtain the solution to larger subproblems.
    - We **sort the subproblems by "size"** and **solve them in size order**, **"smallest" first**.
    - When we first see a subproblem, we have already solved all of its prerequisite subproblems.
  - The results are stored in a table.

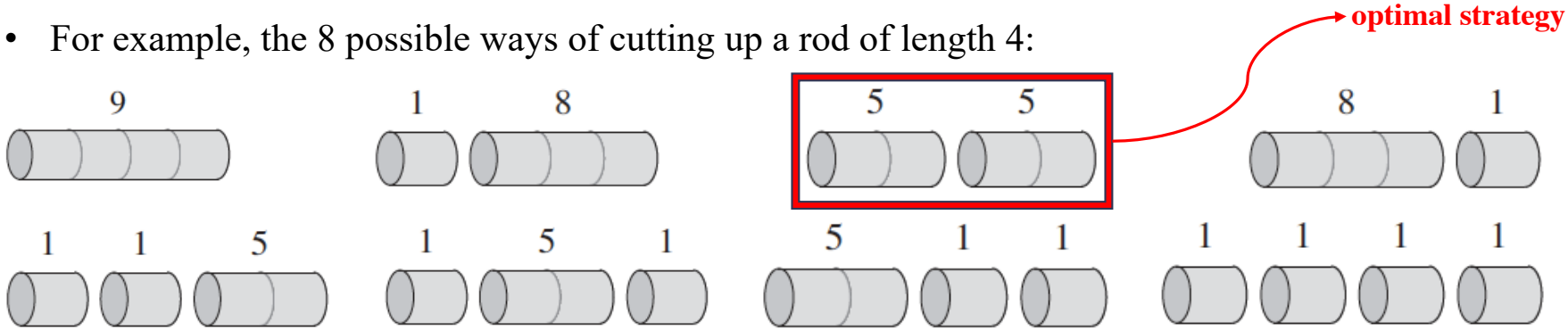
Rod cutting

- Serling Enterprises **buys long steel rods** and **cuts them into shorter rods**, which it **then sells**. Each cut is free.
  - The management wants to know the **best way to cut up the rods**.

❖ **Given a rod of length n inches** and **a table of prices p_i for i**, **determine the maximum revenue r_n** obtainable by cutting up the rod and selling the pieces.
A sample price table:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- For example, the 8 possible ways of cutting up a rod of length 4:



Maximum revenue

- Denote a decomposition into pieces using ordinary additive notation:
  - 7=2+2+3 indicates a rod of length 7 is cut into three pieces, two of length 2 and one of 3.
- If an optimal solution cuts the rod into k pieces, for some $1 \le k \le n$,
  - Then an optimal decomposition $n = i_1 + i_2 + \cdots + i_k$
  - provides maximum corresponding revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$
- For our sample problem,

$$
\begin{aligned}
r_1 &= 1 \quad \text{from solution } 1 = 1 \quad \text{(no cuts)}, \\
r_2 &= 5 \quad \text{from solution } 2 = 2 \quad \text{(no cuts)}, \\
r_3 &= 8 \quad \text{from solution } 3 = 3 \quad \text{(no cuts)}, \\
r_4 &= 10 \quad \text{from solution } 4 = 2 + 2, \\
r_5 &= 13 \quad \text{from solution } 5 = 2 + 3, \\
r_6 &= 17 \quad \text{from solution } 6 = 6 \quad \text{(no cuts)}, \\
r_7 &= 18 \quad \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3, \\
r_8 &= 22 \quad \text{from solution } 8 = 2 + 6, \\
r_9 &= 25 \quad \text{from solution } 9 = 3 + 6, \\
r_{10} &= 30 \quad \text{from solution } 10 = 10 \quad \text{(no cuts)}.
\end{aligned}
$$

A recursive structure

DP Selection Principle: When given a set of feasible options to choose from, try them all and take the best.

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

We **don't know** ahead of time **which value of *i* optimizes revenue**.

- Thus we consider all possible values for i and pick the one maximizes revenue.
- **Once we make the first** cut, we may **consider the two pieces as independent instances** of the rod-cutting problem.
  - The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces.

Optimal substructure: **optimal solutions to a problem** incorporate **optimal solutions to related subproblems**, which we may solve **independently**.

- **We may view every decomposition of a length-*n* rod in this way:**
  - as **a first piece** followed by **some decomposition of the remainder**.
  - For the case of no cuts: The first piece has size $i = n$ and revenue p_n, and the remainder has size 0 and revenue $r_0 = 0$.

A simple version of the above equation:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

Recursive implementation for rod cutting

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

- **Input**: An array $p[1 \ldots n]$ of prices and an integer n.
- **Output**: The maximum revenue possible for a rod of length n.

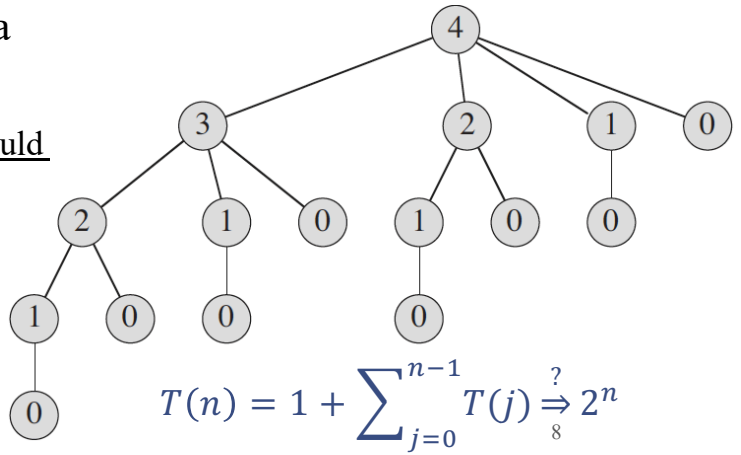
❑ **Prove** by induction on n that this answer is equal to $r_n(?)$, using $r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$

❖ If you were to code up CUT-ROD, your program would take a long time to run.

- You would find that each time you increase n by 1, the running time would approximately double.

➢ Why is CUT-ROD so inefficient?

- **Recursion tree for $n = 4$; it solves same subproblems repeatedly:**
- **Find $T(n)$, the total number of calls made to CUT-ROD?**



$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \overset{?}{\Rightarrow} 2^n$$

The power of DP method

Observed the recursive solution is inefficient because it solves the same subproblems repeatedly.

- We arrange for **each subproblem to be solved only once**, ***saving*** its solution.
  - If we need to refer to this subproblem's solution again later, we can just look it up.
- Dynamic programming thus uses additional memory to save computation time;
  - it serves an example of a ***time-memory trade-off***.

This *saving* may be dramatic:

- An **exponential-time solution** may be transformed into a **polynomial-time solution**.
- Note that **a DP approach runs in polynomial time** when
  - **the number of *distinct* subproblems involved is polynomial and**
  - **we can solve each such subproblem in polynomial time**.

Top-down procedure for rod cutting

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

- Lines1-3 initialize an auxiliary array $\boldsymbol{r[0 \ldots n]}$ with the value $-\infty$ to denote "**unknown**". (Known revenue values are always nonnegative.)
- Line 4 calls the memoized version of our procedure.

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

- Line 1 **checks to see whether the desired value is already known**, and **if it is** then line 2 returns it.
- **Otherwise**, lines 3–7 compute the desired value q in the usual manner, line 8 saves it in $\boldsymbol{r[n]}$, and line 9 returns it.

Bottom-up procedure for rod cutting

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

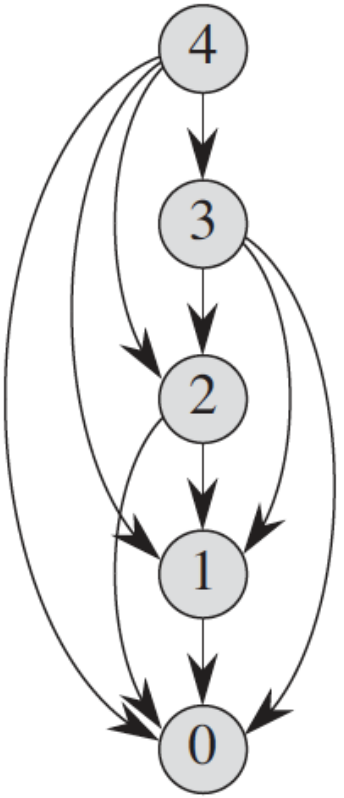
- The bottom-up procedure is simpler.
- Line 2 initializes $r[0]$ to 0, since a rod of length 0 earns no revenue.
- Lines 3-6 solve each subproblem of size j, for $j = 1, \ldots, n$, in order of increasing size.
- Line 7 saves in $r[j]$ the solution to the subproblem of size j.

- **Running time**: #iterations of inner for loop in lines 5–6 forms an arithmetic series.
- **Top-down procedure running time**:
  - Solving a subproblem of size n, the for loop in lines 6–7 iterates n times.
  - The total number of iterations of this for loop, over all recursive calls, forms an arithmetic series.

➢ The bottom-up and top-down procedures have the same asymptotic running time $\theta(n^2)$.

Subproblem graph

- A “reduced” or “collapsed” version of the recursion tree.
  - It helps to **understand the set of subproblems involved**, and **how** subproblems **depend on one another**.
  - For example, in this graph, there is a directed edge from x to y: if an optimal solution for x involves an optimal solution for y.
- The time complexity of a subproblem:
  - It is **proportional to the degree** (number of outgoing edges) of the corresponding vertex
- The number of subproblems:
  - It is equal to the **number of vertices** in the subproblem graph.
- The **running time of dynamic programming** is **linear** in the **number of vertices and edges**.



Reporting the solution for rod cutting

- Observed we computed the value of an optimal solution.

- Now, we want to report the **actual solution (a choice that led to the optimal value)**.
  - Extended version of BOTTOM-UP-CUT-ROD:
  - For solving a subproblem of size j, in line 8, **we store in $s[j]$, the optimal size ($= i$) of the first piece to cut off**.
  - Print the cuts:

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

Reporting the solution for rod cutting

- Calling EXTENDED-BOTTOM-UP-CUT-ROD with the following array p,

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

  - would return the following two arrays r and s:

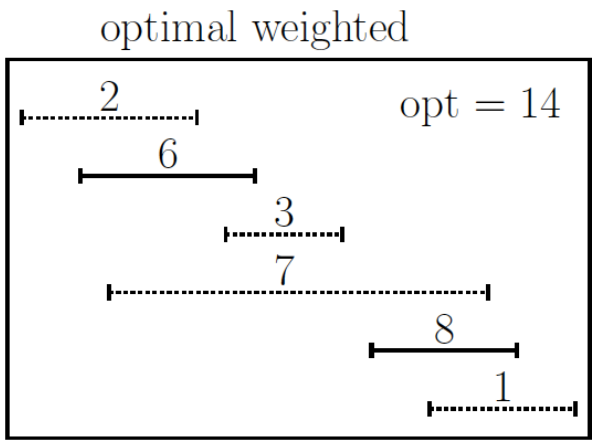
i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

- Calling PRINT-CUT-ROD-SOLUTION(p, n):
  - By n=10, would print just 10, meaning not cut.
  - By n=7, would print two cuts 1 and 6.

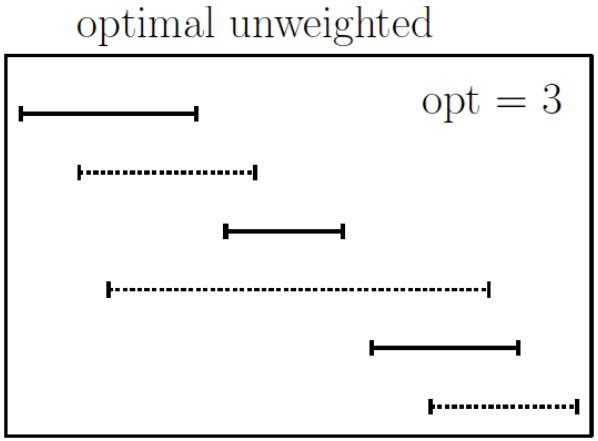
Weighted interval scheduling (WIS)

- Let $S = \{1, \ldots, n\}$ be a set of n activity requests.
  - Each activity starts at time s_i and ends at a time f_i.
  - Each activity has a numeric value (weight) v_i
- Two requests are **compatible** if their intervals do not overlap.

❖ **Weighted interval scheduling (WIS)**: Find a set of compatible requests such that sum of values of the scheduled requests is maximum.

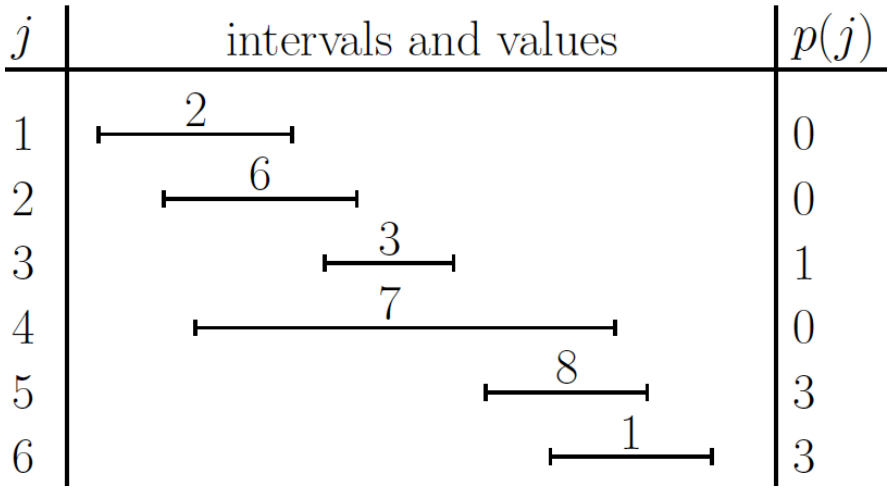


If all activities have the same weights:



A recursive formulation for WIS

- Sort the requests in nondecreasing order of finish time f_i, so that $f_1 \leq \ldots \leq f_n$.
- Define $p(j) = \max\{i : f_i < s_j\}$.



How shall we define the subproblems?

- Define **opt(i)**:
  - **Maximum over the first i requests.**
  - opt(0) = 0.

Compute opt(j) for an arbitrary j:

There are two possibilities:

1. Request j is not in the optimal schedule: **$\mathbf{opt(j) = opt(j-1)}$**
2. Request j is in the optimal schedule: **$\mathbf{opt(j) = v_j + opt(p(j))}$**

Recursive implementation for WIS

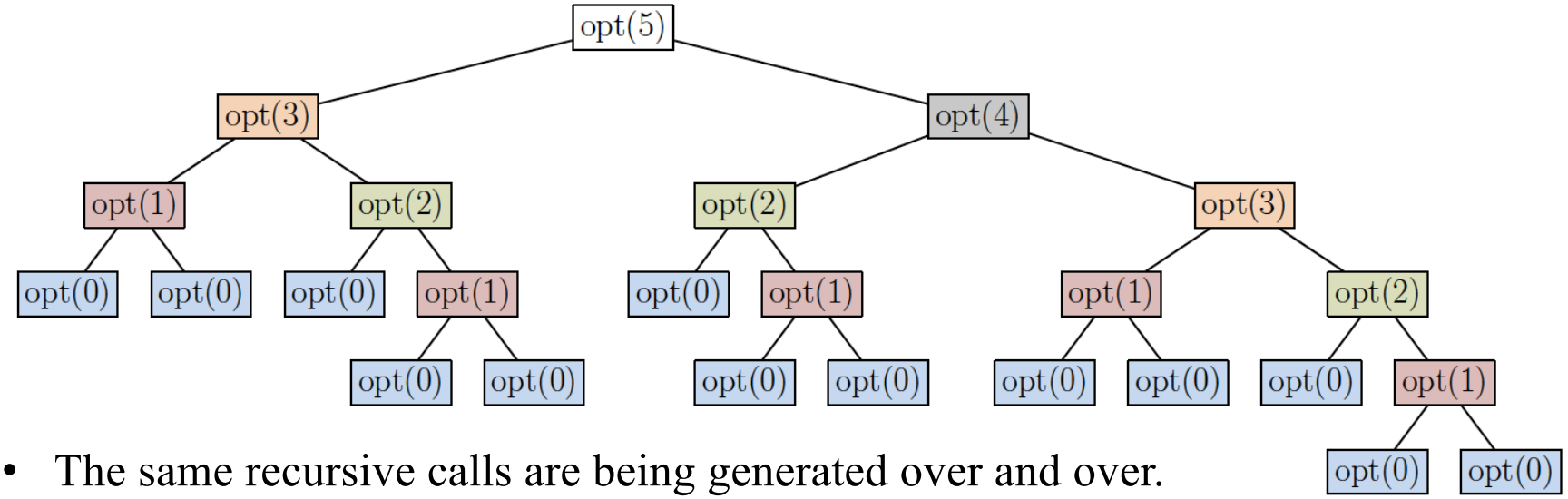
DP Selection Principle: When given a set of feasible options to choose from, try them all and take the best.

- Thus, we have $\mathbf{opt}(j) = \mathbf{max}(\mathbf{opt}(j-1), vj + \mathbf{opt}(p(j)))$.
- In order to **compute opt(j) optimally**, the **two subproblems** $\mathbf{opt}(j-1)$ and $\mathbf{opt}(p(j))$ **should also be computed optimally**.

```
recursive-WIS(j) {
    if (j == 0) return 0
    else return max( recursive-WIS(j-1),
                     v[j] + recursive-WIS(p[j]) )
}
```

Recursive implementation for WIS

- Suppose that $p(j) = j - 2$, for all j.
  - Let $T(j)$ be the number of recursive calls to opt(0).
  - $T(0) = 1$. $T(1) = T(0) + T(0)$. For $j \geq 2$, $\boldsymbol{T(j) = T(j-1) + T(j-2)}$.



- The same recursive calls are being generated over and over.

j	0	1	2	3	4	5	6	7	8	...	20	30	50
$T(j)$	1	2	3	5	8	13	21	34	55	...	17,711	2,178,309	32,951,280,099

Top-down procedure for WIS

- Once a value has been computed for recursive-WIS(j):
  - We store (such optimal) values in a global array $M[1 \ldots n]$.
    - Initialize all the $M[j]$ entries to -1.
    - Use this to determine whether an entry has already been computed.
  - All future attempts access this array rather than making a recursive call.

```
memoized-WIS(j) {
    if (j == 0) return 0
    else if (M[j] has been computed)
         return M[j]
    else {
        M[j] = max( memoized-WIS(j-1),
                    v[j] + memoized-WIS(p[j]) )
        return M[j]
    }
}
```

Running time?

- Memoized-WIS either returns in O(1) time, or it computes one new entry of M.
- Thus memoized version runs in $O(n)$ time.

Bottom-up procedure for WIS

- Simply fill the table up, one entry at a time!

```
bottom-up-WIS() {
    M[0] = 0
    for (j = 1 to n) {
        if (M[j-1] > v[j] + M[p[j]] ) {
            M[j] = M[j-1];
            pred[j] = j-1;
        }
        else {
            M[j] = v[j] + M[p[j]];
            pred[j] = p[j];
        }
    }
}
```

- How do we compute the schedule itself?
- Most DP formulations focus on computing the numeric optimal value.

For computing the final schedule:

- save a predecessor pointer (pred[j])
- which reminds of which choice we made ($j - 1$ or $p(j)$).

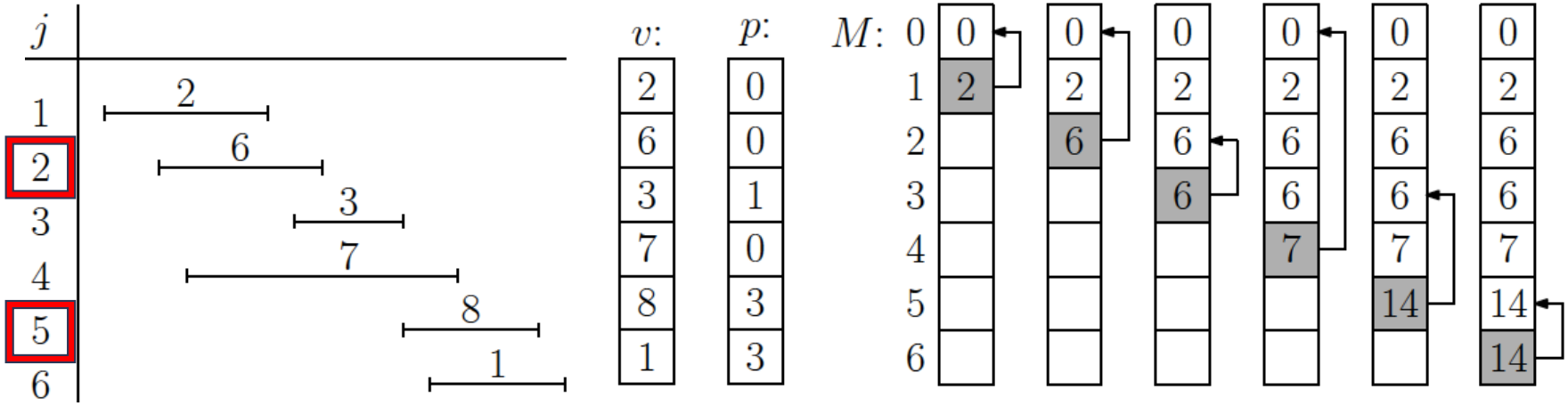
- Running time: $O(n)$
- Would the algorithm be correct if, rather than sorting the requests by *finish time*, we had instead sorted them by *start time*?

Computing the Final Schedule

```
get-schedule() {
    j = n
    sched = (empty list)
    while (j > 0) {
        if (pred[j] == p[j]) {
            prepend j to the front of sched
        }
        j = pred[j]
    }
}
```

- We know that the value of M[j] arose from two distinct possibilities:
  1. We **didn't** take j and just used the result of M[$j - 1$], or
  2. We **did** take j, added its value v_j, and used M[$p(j)$].

Computing the Final Schedule



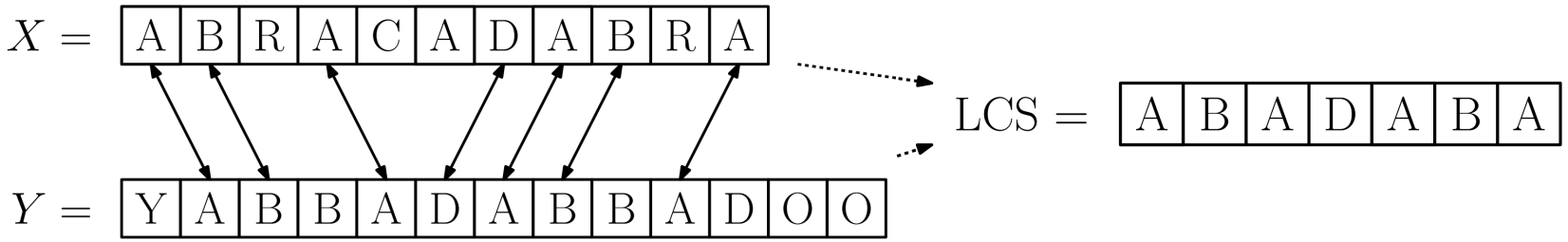
Following the predecessor pointers back from M[n]:

- If we arrive at M[j], we check whether pred[j] = p[j]:
  - If so, we can surmise that we did used the jth request.
  - In the above example, what are the requests that are in the final schedule?
    - **5** and **2**, and the final optimal value is 14.

Longest common subsequence (LCS)

- **Motivation**: Determining the degree of similarity between two strings.
  - We can compute the lengths of their *longest common subsequence*.
- **Definition**: Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Z = \langle z_1, z_2, \ldots, z_k \rangle$, Z is a **subsequence** of X if there are k indices $\langle i_1, i_2, \ldots, i_k \rangle$ $(1 \le i_1 < i_2 < \cdots < i_k \le n)$ such that $Z = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_k} \rangle$.

❖ **Longest Common Subsequence (LCS):** Given two strings X and Y, the **longest common subsequence** of X and Y is a ***longest sequence Z*** that is a ***subsequence of X and Y***.



A recursive formulation for LCS

We break the problem into smaller pieces by

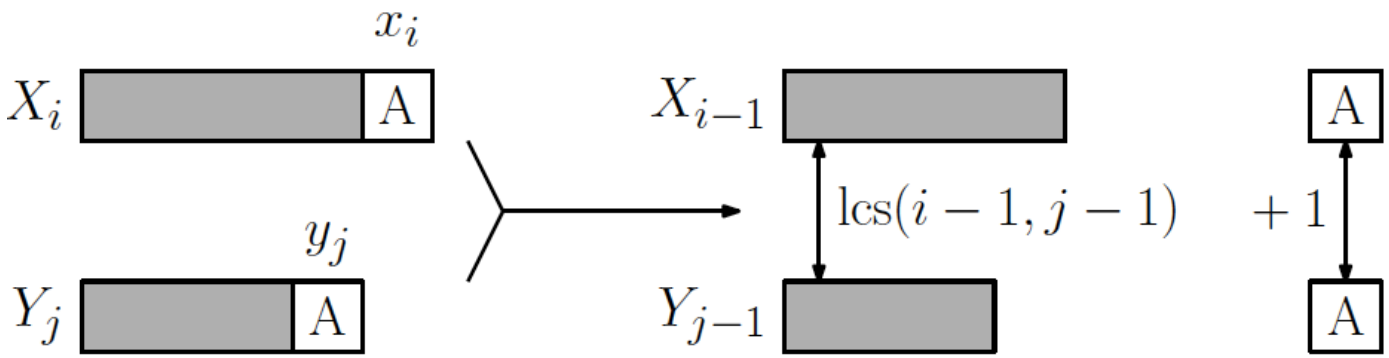
- considering all pairs of prefixes, and
- computing the LCS for every possible pair of prefixes.

- Let $\boldsymbol{X_i} = \langle x_1, x_2, \ldots, x_i \rangle$ and $\boldsymbol{Y_j} = \langle y_1, y_2, \ldots, y_j \rangle$; X_0 and Y_0 are the empty sequences.
- Let $\boldsymbol{lcs(i,j)}$ denote **the length of the LCS of $\boldsymbol{X_i}$ and $\boldsymbol{Y_j}$.**
- Example: $X_5 = ABRAC$ and $Y_6 = YABBAD$. Then $lcs(5,6) = 3$.

Computing $lcs(i, j)$

A recursive formulation for computing $lcs(i, j)$:

1. **Basis**: $lcs(i, 0) = lcs(j, 0) = 0$.
2. **Last characters match**:
   - If $x_i = y_j$, then
   - $lcs(i, j) = lcs(i - 1, j - 1) + 1$.

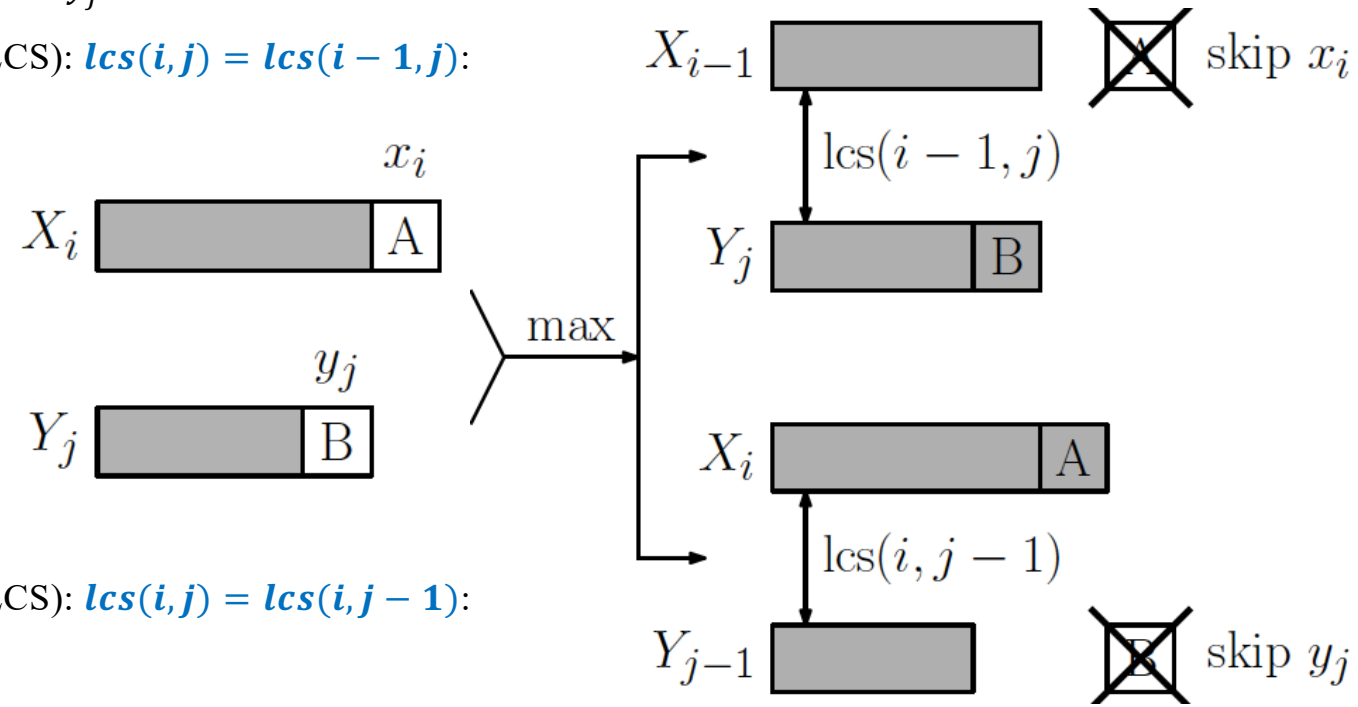


Computing $lcs(i, j)$

3. Last characters do not match:

- In this case x_i and y_j cannot both be in the LCS.

Option 1: ($x_i \notin$ LCS): $\boldsymbol{lcs(i,j) = lcs(i-1,j)}$:



Option 2: ($y_j \notin$ LCS): $\boldsymbol{lcs(i,j) = lcs(i,j-1)}$:

DP Selection Principle: When given a set of feasible options to choose from, try them all and take the best.

A recursive formulation for $lcs(i, j)$

Combining 1,2, and 3, we have:

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

- A direct recursive implementation of this rule will be very inefficient.

➢ Next, we provide two efficient DP procedures.

- Create a 2-dimensional array $\text{lcs}[0..m, 0..n]$, where $m = |X|$ and $n = |Y|$.

Top-down procedure for LCS

```
memoized-lcs(i,j) {
    if (lcs[i,j] has not yet been computed) {
        if (i == 0 || j == 0)
            lcs[i,j] = 0
        else if (x[i] == y[j])
            lcs[i,j] = memoized-lcs(i-1, j-1) +1
        else
            lcs[i,j] = max(memoized-lcs(i-1, j), memoized-lcs(i, j-1))
    }
    return lcs[i,j]
}
```

- Running time:
  - Each time we call memoized-lcs(i, j), if it computed it returns in $O(1)$ time.
  - Each call to memoized-lcs(i, j) generates a $O(1)$ additional calls.
  - Total running time is $O(mn)$.

Bottom-up procedure for LCS

```
bottom-up-lcs() {
    lcs = new array [0..m, 0..n]
    for (i = 0 to m) lcs[i,0] = 0
    for (j = 0 to n) lcs[0,j] = 0
    for (i = 1 to m) {
        for (j = 1 to n) {
            if (x[i] == y[j])
                lcs[i,j] = lcs[i-1, j-1] + 1
            else
                lcs[i,j] = max(lcs[i-1, j],
        }                          lcs[i, j-1])
    }
    return lcs[m, n]
}
```

- Compute lcs array for $X = \langle BACDB \rangle$ and $Y = \langle BDCB \rangle$?

		0	1	2	3	$4 = n$
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
$m = 5$	B	0	1	2	2	3

- The running time and space is $O(mn)$.

Saving hints for extracting the LCS

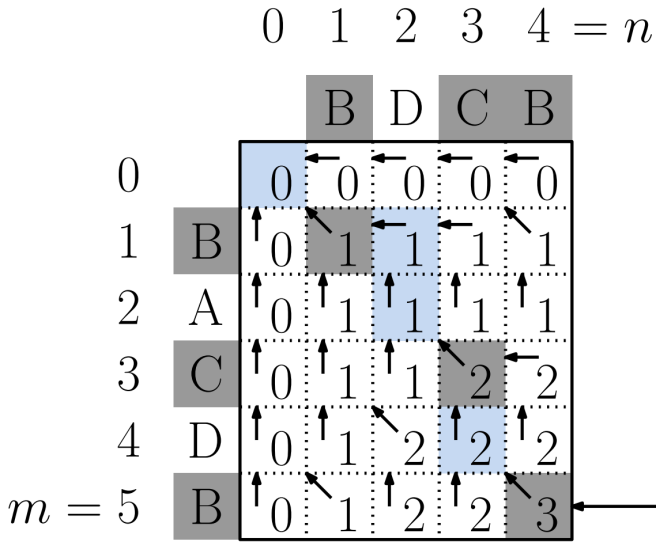
```
bottom-up-lcs-with-hints() {
    lcs = new array [0..m, 0..n]
    h = new array [0..m, 0..n]
    for (i = 0 to m) { lcs[i,0] = 0;  h[i,0] = skipX }
    for (j = 0 to n) { lcs[0,j] = 0;  h[0,j] = skipY }
    for (i = 1 to m) {
        for (j = 1 to n) {
            if (x[i] == y[j])
                { lcs[i,j] = lcs[i-1, j-1] + 1;  h[i,j] = addXY }
            else if (lcs[i-1, j] >= lcs[i, j-1])
                { lcs[i,j] = lcs[i-1, j];  h[i,j] = skipX }
            else
                { lcs[i,j] = lcs[i, j-1];  h[i,j] = skipY }
        }
    }
    return lcs[m, n]
}
```

Extracting the LCS

- **$\mathbf{skip_X}$ (‘↑’)**: Do not include x_i to the LCS and continue with $lcs[i-1, j]$.
- **$\mathbf{skip_Y}$ (‘←’)**: Do not include y_j to the LCS and continue with $lcs[i, j-1]$.
- **$\mathbf{add_{XY}}$ (‘↖’)**: Add $x_i (= y_j)$ to the LCS and continue with $lcs[i-1, j-1]$.

How do we use the hints to reconstruct the answer?

1. Start at the last entry.
2. If $\boldsymbol{h[i,j] = \mathbf{add_{XY}}}$, we know that $\boldsymbol{x_i (= y_j)}$ **is appended to the LCS**
   - Continue with entry $[i-1, j-1]$.
3. If $\boldsymbol{h[i,j] = \mathbf{skip_X}}$ we know that $\boldsymbol{x_i}$ **is not in the LCS**:
   - Continue with entry $[i-1, j]$.
4. If $\boldsymbol{h[i,j] = \mathbf{skip_Y}}$ we know that $\boldsymbol{y_j}$ **is not in the LCS**:
   - Continue with entry $[i, j-1]$.

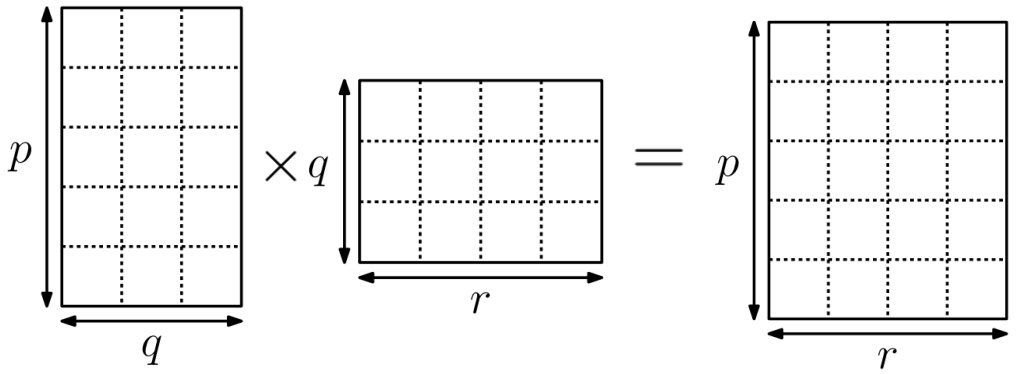


Extracting the LCS

```
get-lcs-sequence() {
    LCS = new empty character sequence
    i = m; j = n
    while(i != 0 or j != 0)
        switch h[i,j]
            case addXY:
                prepend x[i] to front of LCS
                i--;  j--;    break
            case skipX: i--;  break
            case skipY: j--;  break
    return LCS
}
```

Chain matrix multiplication

- **Motivation**: In compiler design (for code optimization) and in databases (for query optimization):
  - Determine the optimal sequence for performing a series of operations.
- Suppose that we wish to multiply a series of matrices $C = A_1 \cdot A_2 \cdot \ldots \cdot A_n$
- We can parenthesize the above multiplication however we like.
- When multiplying a matrix $A_{p\times q}$ times a matrix $B_{q\times r}$, the result will be a $\boldsymbol{p\times r}$ **matrix C**.



$$C[i,j] = \sum_{k=1}^{q} A[i,k] \cdot B[k,j]$$

multiplication time $= O(pqr)$

Chain matrix multiplication

- **Any legal "parenthesization"** will lead to a **valid result**, but **not all involve the same number of operations**.
- **Example**: Consider the 3 matrices $A_1(5\times4)$, $A_2(4\times6)$ and A_3 (6×2).

$$\begin{aligned}\text{cost}[((A_1A_2)A_3)] &= (5\cdot4\cdot6)+(5\cdot6\cdot2)=180,\\ \text{cost}[(A_1(A_2A_3))] &= (4\cdot6\cdot2)+(5\cdot4\cdot2)=88.\end{aligned}$$

❖ **Chain Matrix Multiplication (CMM):** Given a sequence of matrices $A_1, \ldots, A_n$ and dimensions $p_0, \ldots, p_n$ where A_i is of dimension $p_{i-1}\times p_i$. **Determine the order of multiplication** that **minimizes the number of operations**.

- Note that we do not want to perform the multiplications.

A recursive formulation for CMM

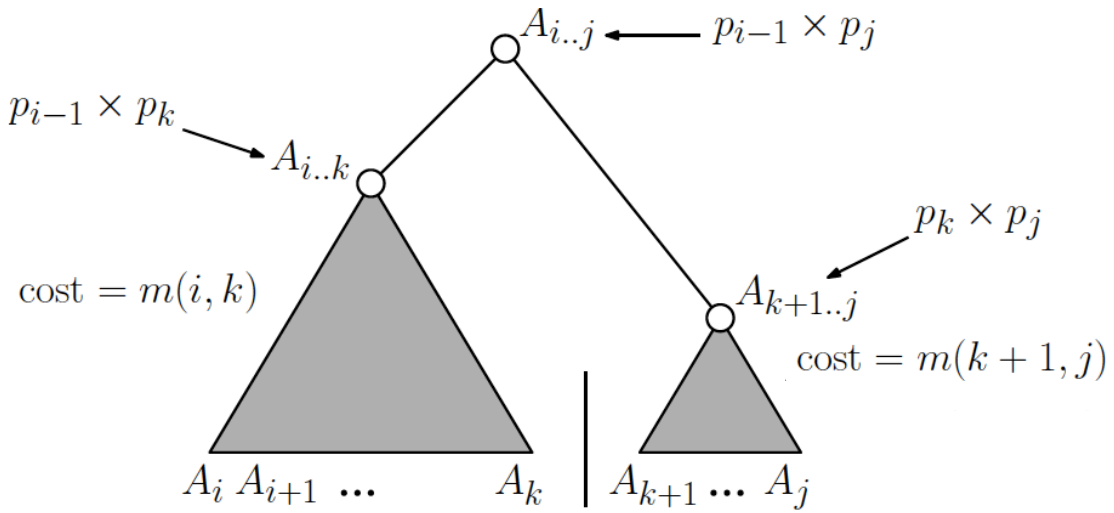
- Break the problem into subproblems, whose solutions can be combined to solve the global problem:
- Let $A_{i\ldots j}$ denote the result of **multiplying matrices i through j**.
- Consider the highest level of parenthesization: $A_{1\ldots n} = A_{1\ldots k} \cdot A_{k+1\ldots n}$ for $1 \leq k \leq n-1$.
- We need to decide:
  - **What is the best place to split the chain (what is k)?**
  - **How do we parenthesize each of the two subsequences $A_{1\ldots k}$ and $A_{k+1\ldots n}$?**

➢ **DP Selection Principle: We will try all possible choices of k and take the best of them.**

A recursive formulation for CMM

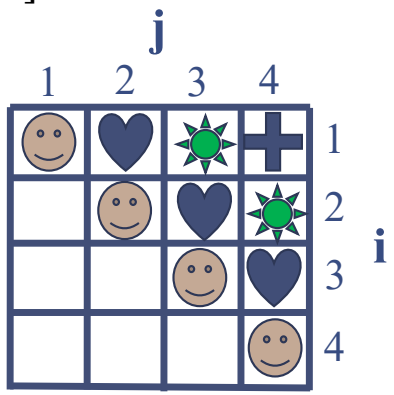
Let $\boldsymbol{m(i,j)}$ denote the **minimum number of multiplications to compute** $\boldsymbol{A_{i..j}}$, $1 \le i \le j \le n$.

- **If $\boldsymbol{i = j}$, $\boldsymbol{m(i,i) = 0}$.**
- **Else (for $\boldsymbol{i < j}$)**, **split** $\boldsymbol{A_{i..j}}$ into two groups $\boldsymbol{A_{i..k}}$ and $\boldsymbol{A_{k+1..j}}$ (by considering each k, $i \le k < j$).
  - $\boldsymbol{m(i,j) = \min_{i \le k < j}(m(i,k) + m(k+1,j) + p_{i-1}p_k p_j)}$
  - Note that the time to multiply $A_{i..k}$ and $A_{k+1..j}$ is $\boldsymbol{p_{i-1}p_k p_j}$, which is the **last multiplication cost.**



Bottom-up procedure for CMM

- Store the values of $m(i, j)$ in a 2-dimensional array $m[1 \ldots n, 1 \ldots n]$.
  - The trickiest part of the process is arranging the order in which to compute the values.
- We ***cannot*** just compute the matrix in the simple ***row-by-row***:
  - **Example**: When computing $m[3, 5]$, we need $m[3, 4]$ & $m[4, 5]$, but $m[4, 5]$ is in row 4.
- The trick is to compute the matrix ***diagonal-by-diagonal***.
- **We solve the problem,**
  - First for chains of **length 1** (which is trivial),
  - then for chains of **length 2**, and **so on**,
  - until we **come to $m[1, n]$** (which is the total chain of length n).



matrix *m*

Bottom-up procedure for CMM

```
Matrix-Chain(p[0..n]) {
    s = array[1..n-1, 2..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] = INFINITY
            for (k = i to j - 1) {
                cost = m[i, k] + m[k+1, j] + p[i-1]*p[k]*p[j]
                if (cost < m[i, j]) {
                    m[i, j] = cost
                    s[i, j] = k
                }
            }
        }
    }
    return m[1, n] (final cost) and s (splitting markers)
}
```

min … $i \le k < j$

- Let $L = j - i + 1$ denote the **length of the subchain** for $1 \le i \le j \le n$.
  - ➢ **Outer loop runs from $L = 2, \dots, n$.**
  - ➢ **Inner loop runs on all values of i: since $j \le n$, $i \le n - L + 1$.**

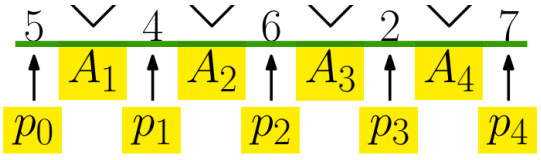
✓ The running time is $O(n^3)$.

Extracting the final Sequence

- The basic idea is to leave a **split marker**, indicating what the best split is.
  - The value of k that leads to the minimum value of $m[i,j]$.
  - Intuitively, $s[i,j]$ tells us what multiplication to perform last.
- Suppose that $s[i,j] = k$; this tells us that the best way for computing $A_{i\ldots j}$ is:
  - First multiply the subchain $A_{i\ldots k}$.
  - Then multiply the subchain $A_{k+1\ldots n}$.
  - Finally multiply these together.

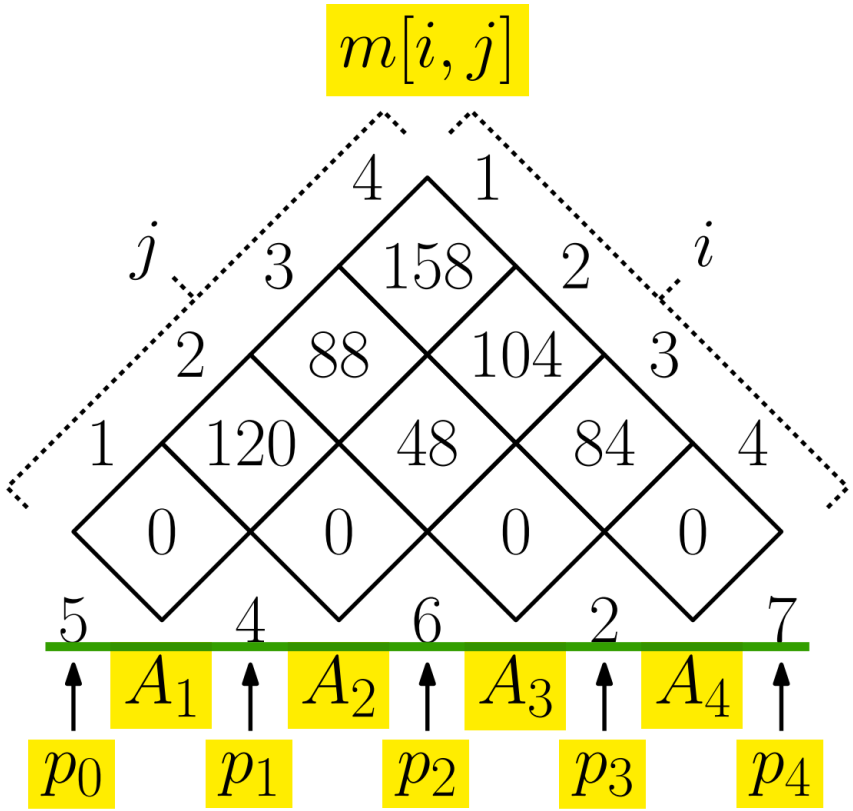
❑ Procedure for extracting final order:

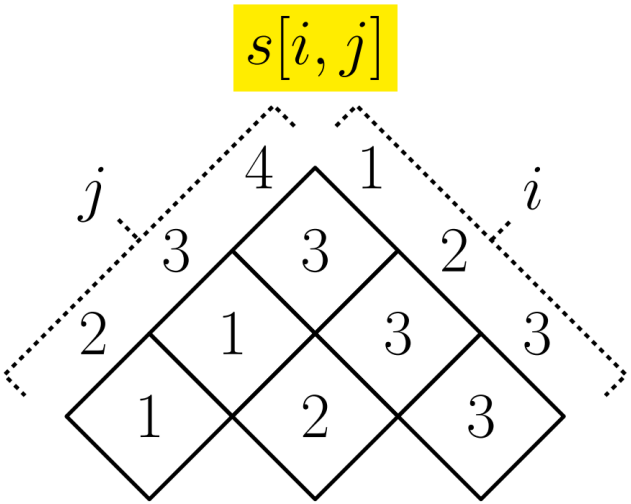
- An example:

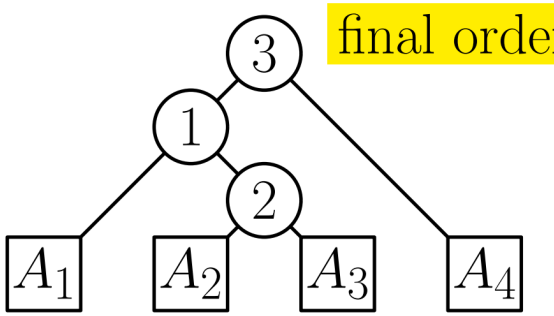



```
do-mult(i, j) {
    if (i == j)
        return A[i]
    else {
        k = s[i,j]
        X = do-mult(i, k)
        Y = do-mult(k+1, j)
        return X * Y
    }
}
```

An example


$m[i,j]$
4
1
j
3
158
2
i
2
88
104
3
1
120
48
84
4
0
0
0
0
5
4
6
2
7
A_1
A_2
A_3
A_4
p_0
p_1
p_2
p_3
p_4


$s[i,j]$
j
4
1
i
3
3
2
2
1
3
3
1
2
3


final order
3
1
2
A_1
A_2
A_3
A_4