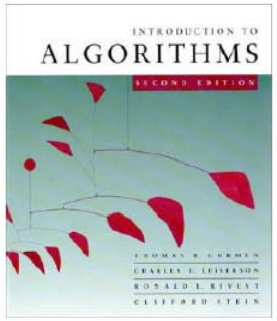
Introduction to Algorithms



Chapter 15: Dynamic Programming

Dynamic Programming

- Well known algorithm design techniques:
  - Brute-Force (iterative) algorithms
  - Divide-and-conquer algorithms
- Another strategy for designing algorithms is **dynamic programming**.
  - Used when problem breaks down into recurring small subproblems
- ***Dynamic programming*** is typically applied to optimization problems.
  - In such problem there can be ***many solutions***. Each solution has a value, and we wish to find ***a solution*** with the optimal value.

Divide-and-conquer

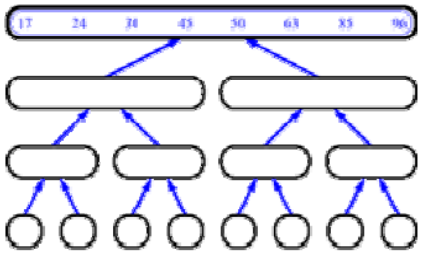
- **Divide-and-conquer** method for algorithm design:
- **Divide**: If the input size is too large to deal with in a straightforward manner, divide the problem into two or more disjoint subproblems
- **Conquer**: conquer recursively to solve the subproblems
- **Combine**: Take the solutions to the subproblems and "merge" these solutions into a solution for the original problem

Divide-and-conquer - Example

- For example, **MergeSort**

```
Merge-Sort(A, p, r)
   if p < r then
      q←(p+r)/2
      Merge-Sort(A, p, q)
      Merge-Sort(A, q+1, r)
      Merge(A, p, q, r)
```

- The **subproblems** are **independent**, **all different.**



Dynamic programming

- **Dynamic programming** is a way of improving on **inefficient** *divide-and-conquer* algorithms.
- By "*inefficient*", we mean that ***the same recursive call is made over and over***.
- If **same** *subproblem* is solved several times, we can use **table** to store result of a subproblem the first time it is computed and thus never have to recompute it again.
- Dynamic programming is applicable when the subproblems are dependent, that is, when subproblems share subsubproblems.
- **"Programming"** refers to a tabular method

Difference between DP and Divide-and-Conquer

- Using Divide-and-Conquer to solve these problems is **inefficient** because the **same** common **subproblems** have to be solved **many times**.
- DP will solve each of them **once** and **their answers are stored in a table** for future use.

Elements of Dynamic Programming (DP)

DP is used to solve problems with the following characteristics:

- Simple subproblems
  - We should be able to break the original problem to **smaller subproblems** that have the **same** structure
- Optimal substructure of the problems
  - The **optimal solution** to the problem contains within **optimal solutions to** its **subproblems**.
- Overlapping sub-problems
  - there exist some places where we solve the **same subproblem** more than **once**.

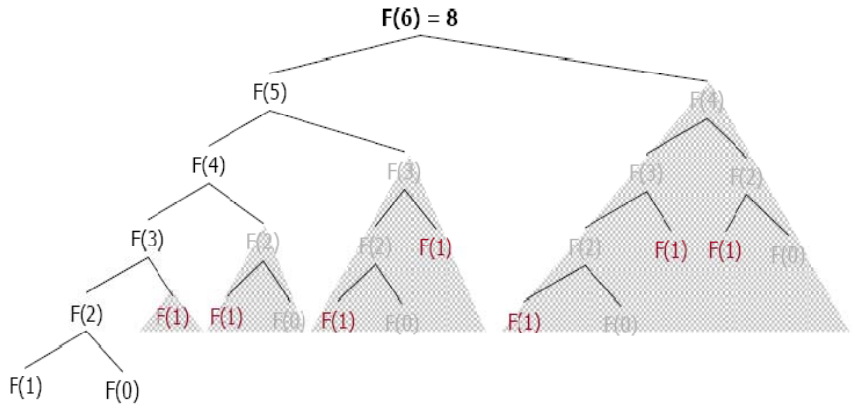
Steps to Designing a Dynamic Programming Algorithm

1. Characterize optimal substructure
2. Recursively define the value of an optimal solution
3. Compute the value bottom up
4. (if needed) Construct an optimal solution

Fibonacci Numbers

- $Fn = F_{n-1} + F_{n-2}$ $\quad n \geq 2$
- $F_0 = 0,\ F_1 = 1$
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
- Straightforward recursive procedure is slow!
- Let's draw the recursion tree

Fibonacci Numbers



- We keep calculating the same value over and over!

Fibonacci Numbers

- We can calculate *Fn* in ***linear* time** by remembering solutions to the solved subproblems – *dynamic programming*
- Compute solution in a bottom-up fashion
- In this case, only two values need to be remembered at any time

```
Fibonacci(n)
    F0←0
    F1←1
    for i ← 2 to n do
        Fi ← Fi-1 + Fi-2
```

Ex1:Assembly-line scheduling

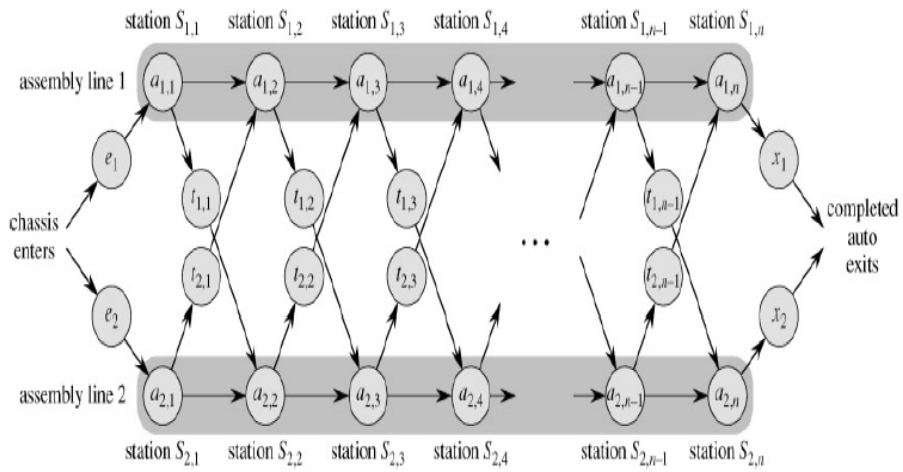
- Cars factory with two assembly lines.
  - Each line has the same number "n" of stations. Numbered j = 1, 2, ..., n.
  - We denote the jth station on line i (where i is 1 or 2) by $S_{i,j}$.
  - The jth station on line 1 ($S_{1,j}$) performs the same function as the jth station on line 2 ($S_{2,j}$).
  - The time required at each station varies, even between stations at the same position on the two different lines, as each assembly line has different technology.
  - time required at station $S_{i,j}$ is ($a_{i,j}$) .
  - There is also an entry time (e_i) for the chassis to enter assembly line i and an exit time (x_i) for the completed auto to exit assembly line i.

Ex1:Assembly-line scheduling

- After going through station $S_{i,j}$, *can either*
  - q stay on same line
    - next station is $S_{i,j+1}$
    - no transfer cost , or
  - q **transfer to other line**
    - next station is $S_{3-i,j+1}$
    - transfer cost from $S_{i,j}$ *to* $S_{3-i,j+1}$ *is* $t_{i,j}$ *(j = 1, … , n–1)*
    - No $t_{i,n}$ *because the assembly line is complete after* $S_{i,n}$

Ex1:Assembly-line scheduling

•(Time between adjacent stations are nearly 0).



Problem Definition

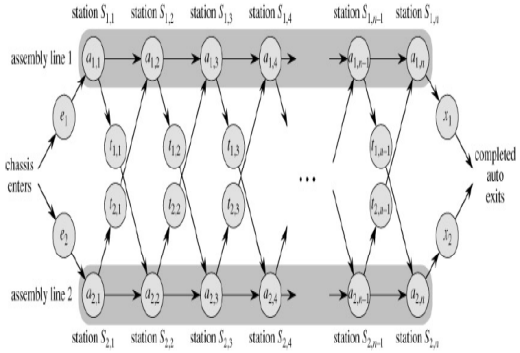
- **Problem:** Given all these costs, what stations should be chosen from line 1 and from line 2 for **minimizing** the total time for car assembly.
- "Brute force" is to try all possibilities.
  - requires to examine $\Omega(2^n)$ possibilities
  - Trying all $\mathbf{2^n}$ **subsets** is infeasible when n is large.
  - Simple example : 2 stations → (2^n) possibilities =4

Step 1: Optimal Solution Structure

- **optimal substructure** : choosing the best path to S_{ij}.
- The structure of the fastest way through the factory (from the starting point)
- The fastest possible way to get through $S_{i,1}$ (i = 1, 2)
  - Only one way: from entry starting point to $S_{i,1}$
  - take time is entry time (e_i)

Step 1: Optimal Solution Structure

- The fastest possible way to get through $S_{i,j}$ (i = 1, 2) (j = 2, 3, ..., n). Two choices:
  - Stay in the same line: $S_{i,j-1} \rightarrow S_{i,j}$
  - Time is $T_{i,j-1} + a_{i,j}$
    - If the fastest way through $S_{i,j}$ is through $S_{i,j-1}$, it must have taken a fastest way through $S_{i,j-1}$
  - Transfer to other line: $S_{3-i,j-1} \rightarrow S_{i,j}$
  - Time is $T_{3-i,j-1} + t_{3-i,j-1} + a_{i,j}$
    - Same as above

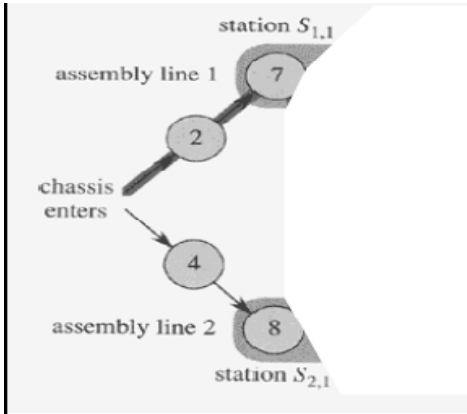


Step 1: Optimal Solution Structure

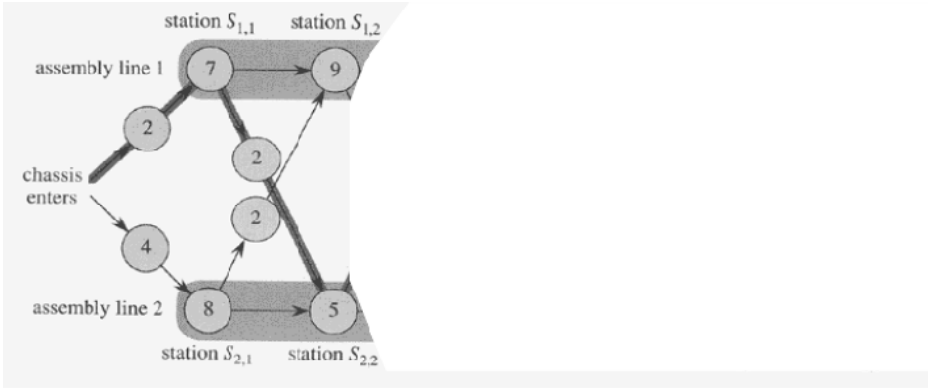
- An optimal solution to a problem
  - finding the fastest way to get through $S_{i,j}$
- contains within it an optimal solution to sub-problems
  - finding the fastest way to get through either $S_{i,j-1}$ or $S_{3-i,j-1}$
- Fastest way from starting point to $S_{i,j}$ is either:
  - The fastest way from starting point to $S_{i,j-1}$ and then directly from $S_{i,j-1}$ to $S_{i,j}$

  or
  - The fastest way from starting point to $S_{3-i,j-1}$ then a transfer from line 3-i to line i and finally to $S_{i,j}$
- **→Optimal Substructure.**

Example



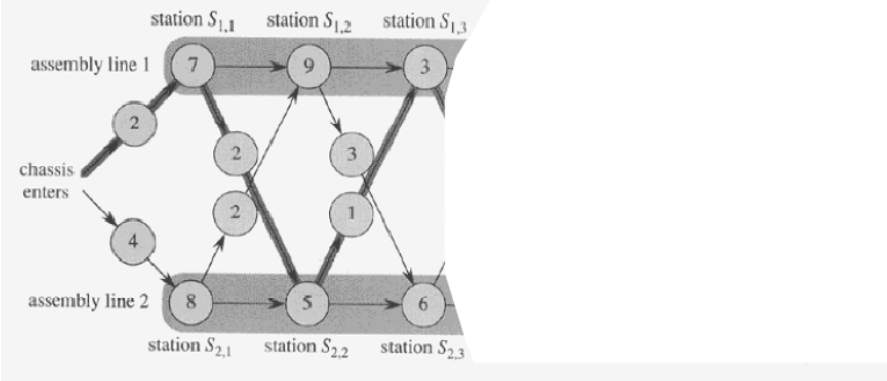
S1,1 ➔ 2 + 7 = 9
S2,1 ➔ 4 + 8 = 12



$S_{1,1}$ ➔ 2 + 7 = 9
$S_{2,1}$ ➔ 4 + 8 = 12

$$S_{1,2} = \begin{cases} S_{1,1} + 9 = 9 + 9 = 18 \\ S_{2,1} + 2 + 9 = 12 + 2 + 9 = 23 \end{cases}$$

$$S_{2,2} = \begin{cases} S_{1,1} + 2 + 5 = 9 + 2 + 5 = 16 \\ S_{2,1} + 5 = 12 + 5 = 17 \end{cases}$$



$S_{1,2} \rightarrow 18$
$S_{2,2} \rightarrow 16$

$$S_{1,3} = \begin{cases} S_{1,2} + 3 = 18 + 3 = 21 \\ S_{2,2} + 1 + 3 = 16 + 1 + 3 = 20 \end{cases}$$

$$S_{2,3} = \begin{cases} S_{1,2} + 3 + 6 = 18 + 3 + 6 = 27 \\ S_{2,2} + 6 = 16 + 6 = 22 \end{cases}$$

Step 2: Recursive Solution

- Define the value of an optimal solution **recursively** in terms of the optimal solution to sub-problems
- Sub-problem here
  - finding the fastest way through station j on both lines (i=1,2)
  - Let $f_i[j]$ be the fastest possible time to go from starting point through $S_{i,j}$
- The fastest time to go all the way through the factory: f*

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$$

- x_1 and x_2 are the exit times from lines 1 and 2, respectively

Step 2: Recursive Solution

- The fastest time to go through $S_{i,j}$
  - e_1 and e_2 are the entry times for lines 1 and 2

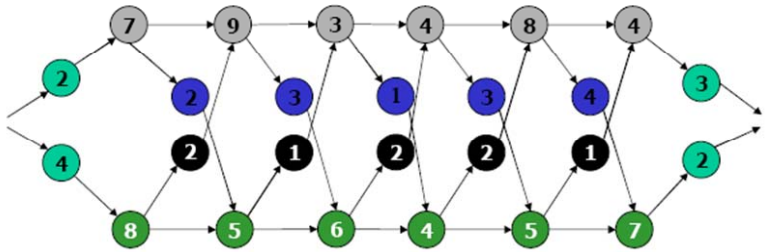
$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1, \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1, \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

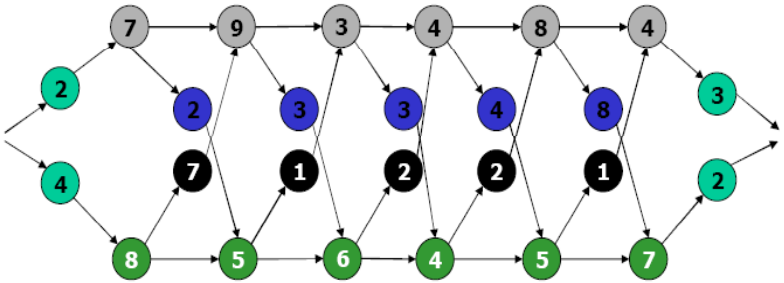
Example

j	1	2	3	4	5	6
a_1	7	9	3	4	8	4
t_2	2	3	1	3	4	
t_1	2	1	2	2	1	
a_2	8	5	6	4	5	7

e_1	2
e_2	4
x_1	3
x_2	2



Example



j	1	2	3	4	5	6
f_1	9	18	20	24	32	35
f_2	12	16	22	25	30	37

Min finish time
$f^* = 35+3 = 38$

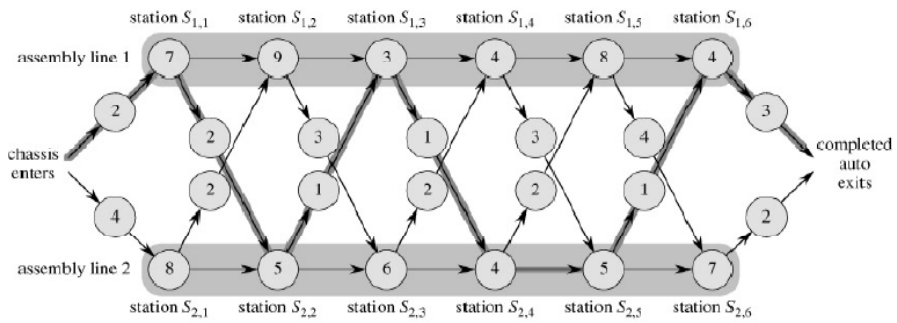
Step 2: Recursive Solution

- To help us keep track of how to construct an optimal solution, let us define
  - $l_i[j]$: line # whose station j-1 is used in a fastest way through $S_{i,j}$ (i = 1, 2, and j = 2, 3,..., n)
- we avoid defining $l_i[1]$ because no station precedes station 1 on either lines.
- We also define
  - l*: the line whose station n is used in a fastest way through the entire factory

Step 2: Recursive Solution

- Using the values of l* and $l_i[j]$ shown in Figure (b) in next slide, we would trace a fastest way through the factory shown in part (a) as follows
  - Starting with $l^* = 1$, use station $S_{1,6}$
  - $l_1[6] = 2$ use station $S_{2,5}$
  - $l_2[5] = 2$ use station $S_{2,4}$
  - $l_2[4] = 1$ use station $S_{1,3}$
  - $l_1[3] = 2$ use station $S_{2,2}$
  - $l_2[2] = 1$ use station $S_{1,1}$
  - The fastest total time comes from choosing stations
  - Line 1: 1, 3, & 6 Line 2: 2, 4, & 5

Step 2: Recursive Solution



(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

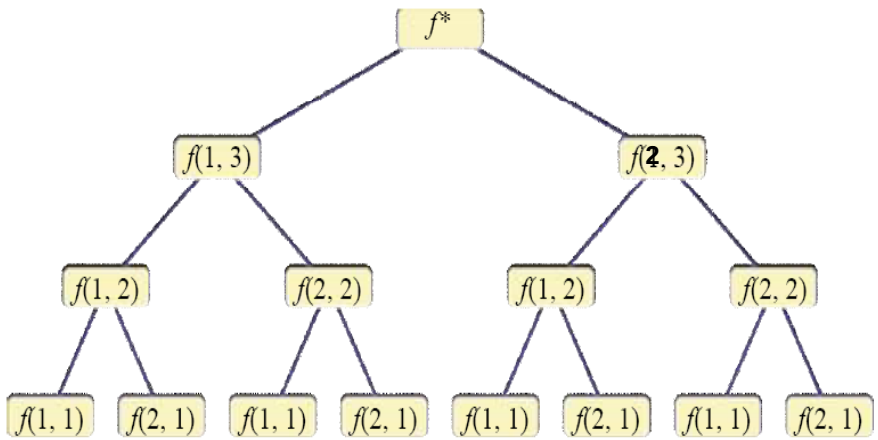
Step 3: Optimal Solution Value

- You can write a divide-and-conquer recursive algorithm to compute f^*.

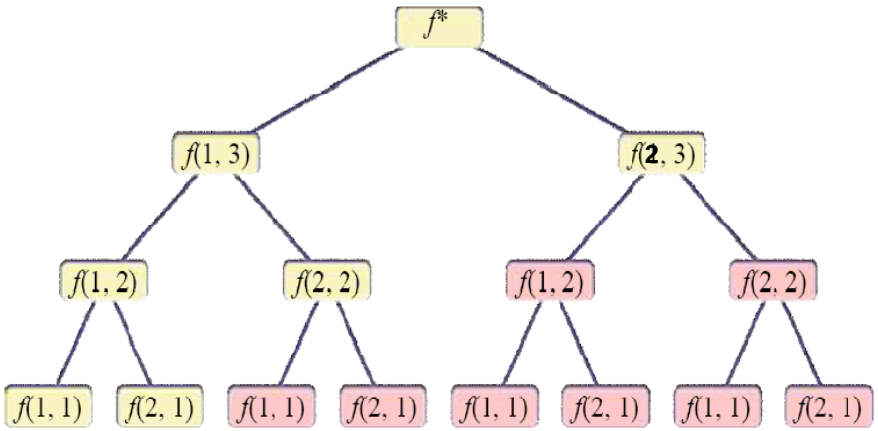
- **Inefficient implementation**

```
public int f(int i, int j) {
   if (j==1) return e[i,1]+a[i,1];
   else {
      int temp1 = f(i, j-1);
      int temp2 = f(3-i, j-1)+t[3-i,j-1];
      return min(temp1,temp2)+a[i,j];
   }
 }
```

Step 3: Optimal Solution Value



Step 3: Optimal Solution Value



Step 3: Optimal Solution Value

- Sketch of proof:
  - Let $T(j)$ be the time taken to execute $f(i, j)$
  - $T(1) = c$ no further recursive calls
  - $T(j) = c + 2\,T(j\text{-}1)$
    - $f(i, j)$ calls both $f(i, j\text{-}1)$ and $f(3\text{-}i, j\text{-}1)$
  - Expand the expression until $T(1)$ …

- So top down isn't a good way to compute $f_i[j]$.

Use substitution method to solve it → $O(2^n)$ → exponential

Step 3: Optimal Solution Value

- We can do much better if we compute the $f_i[j]$ values in different order from the recursive way.
- ***Observation:***
  - $f_i[j]$ depends only on $f_1[j\text{-}1]$ and $f_2[j\text{-}1]$ for $j \geq 2$
- So compute $f_i[j]$ in order of increasing j and store $f_i[j]$ in a table
  - $\Theta(n)$

Step 3: Optimal Solution Value

```
FASTEST-WAY(a, t, e, x, n)
f1[1] ← e1 + a1,1
f2[1] ← e2 + a2,1
for j ← 2 to n
    do if f1[j − 1] + a1,j ≤ f2[j − 1] + t2,j−1 + a1,j
          then f1[j] ← f1[j − 1] + a1,j
               l1[j] ← 1
          else f1[j] ← f2[j − 1] + t2,j−1 + a1,j
               l1[j] ← 2
       if f2[j − 1] + a2,j ≤ f1[j − 1] + t1,j−1 + a2,j
          then f2[j] ← f2[j − 1] + a2,j
               l2[j] ← 2
          else f2[j] ← f1[j − 1] + t1,j−1 + a2,j
               l2[j] ← 1
if f1[n] + x1 ≤ f2[n] + x2
   then f* = f1[n] + x1
        l* = 1
   else f* = f2[n] + x2
        l* = 2
```

Step 3: Optimal Solution Value

```
public int fastestWay() {
   f1[1] = e1+a1[1];
   f2[1] = e2+a2[1];
   for (int j=2; j<=n; j++) {
      f1[j] = min(f1[j-1], f2[j-1] + t2[j-1])
                  + a1[j-1];
      f2[j] = min(f2[j-1], f1[j-1] + t1[j-1])
                  + a2[j-1];
   }
   return min(f1[n]+x1; f2+x2);
}
```

Step 4: Optimal Solution

- Constructing the fastest way through the

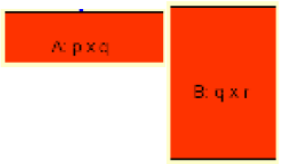
```
PRINT-STATIONS(l, n)
i ← l*
print "line " i ", station " n
for j ← n downto 2
    do i ← li[j]
       print "line " i ", station " j − 1
```

line 1, station 6
line 2, station 5
line 2, station 4
line 1, station 3
line 2, station 2
line 1, station 1

Multiplying Matrices

- Two matrices, *A* with (*p* x *q*) matrix and *B* with (*q* x *r*) matrix, can be multiplied to get *C* with dimensions *p* x *r*, using scalar multiplications

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} = \begin{pmatrix} \dots & \dots & \dots \\ \dots & c_{22} & \dots \\ \dots & \dots & \dots \end{pmatrix} \qquad c_{i,j} = \sum_{l=1}^{m} a_{i,l} \cdot b_{l,j}$$



Multiplying Matrices

```
Matrix-Multiply(A,B):
if columns[A] != rows[B] then
   error "incompatible dimensions"
else for i = 1 to rows[A] do
         for j = 1 to columns[B] do
           C[i,j] = 0
            for k = 1 to columns[A] do
               C[i,j]=C[i,j]+A[i,k]*B[k,j]
return C
```

- Time complexity = $O(pqr)$, where
  - $|A| = p \times q$ and
  - $|B| = q \times r$

Ex2: Matrix-chain multiplication

- We are given a sequence (chain) <A1, A2, ..., An> of n matrices to be multiplied, and we wish to compute the product. A1.A2…An.

- Matrix multiplication is associative, and so all parenthesizations yield the same product,

  (A1 (A2 (A3 A4))),
  (A1 ((A2 A3) A4)),
  ((A1 A2) (A3 A4)),
  ((A1 (A2 A3)) A4),
  (((A1 A2) A3) A4).

Matrix-chain multiplication – cont.

- We can multiply two matrices A and B only if they are **compatible**
  - the number of columns of A must equal the number of rows of B.

- If A is a p × q matrix and B is a q × r matrix, the resulting matrix C is a p × r matrix.

- The time to compute C is dominated by the number of scalar multiplications, which is p q r.

Matrix-chain multiplication – cont.

- Ex: consider the problem of a chain <A1, A2, A3> of three matrices, with the dimensions of 10 × 100, 100 × 5, and 5 × 50, respectively.
- If we multiply according to ((A1 A2) A3), we perform 10 · 100 · 5 = 5000 scalar multiplications to compute the 10 × 5 matrix product A1 A2, plus another 10 · 5 · 50 = 2500 scalar multiplications to multiply this matrix byA3 for a total of 7500 scalar multiplications
- If instead , we multiply as (A1 (A2 A3)), we perform 100 · 5 · 50 = 25,000 scalar multiplications to compute the 100 × 50 matrix product A2 A3, plus another 10 · 100 · 50 = 50,000 scalar multiplications to multiply A1 by this matrix, for a total of 75,000 scalar multiplications.
- Thus, computing the product according to the first parenthesization is 10 times faster.

Matrix-chain multiplication – cont.

- Matrix multiplication is *associative*
  - *(AB)C = A(BC)*
- The **parenthesization matters**
- Consider A X B X C X D, where
  - A is 30x1, B is 1x40, C is 40x10, D is 10x25
- Costs:
  - ((AB)C)D = 1200 + 12000 + 7500 = 20700
  - (AB)(CD) = 1200 + 10000 + 30000 = 41200
  - A((BC)D) = 400 + 250 + 750 = 1400

Matrix Chain Multiplication (MCM) Problem

- Input:
  - Matrices *A*1, *A*2, …, *An*, each *Ai* of size *pi*-1 x *pi*,
- Output:
  - Fully **parenthesised** product *A*1 x *A*2 x … x *An* that **minimizes** the number of scalar multiplications.
- Note:
  - In MCM problem, we are not actually multiplying matrices
  - Our goal is only to determine an order for multiplying matrices that has the lowest cost

Matrix Chain Multiplication (MCM) Problem

- Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications
- So, exhaustively checking all possible parenthesizations does not yield an efficient algorithm

Counting the Number of Parenthesizations

- Denote the number of alternative parenthesizations of a sequence of (n) matrices by P(n) then a fully parenthesized matrix product is given by:

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

(A1 (A2 (A3 A4))) ,
(A1 ((A2 A3) A4)) ,
((A1 A2) (A3 A4)) ,
((A1 (A2 A3)) A4) ,
(((A1 A2) A3) A4).

If n = 4 <A1,A2,A3,A4> then the number of alternative parenthesis is =5
P(4) =P(1) P(4-1) + P(2) P(4-2) + P(3) P(4-3)
= P(1) P(3) + P(2) P(2) + P(3) P(1)= 2+1+2=5
P(3) =P(1) P(2) + P(2) P(1) = 1+1=2
P(2) =P(1) P(1)= 1

Counting the Number of Parenthesizations

Let $T(n)$ be the time complexity of $P(n)$
For $n > 1$, we have

- $T(n) = 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1)$
- Using substitution, we can show that $T(n) \geq 2^{n-1}$
- Hence $T(n) = \Omega(2^n)$

Exhaustively checking all possible parenthesizations take exponential time!

Step 1: Optimal Sub-structure

- Let $A_{i..j}$ $(i \leq j)$ denote the result of $A_i \times A_{i+1} \times \ldots \times A_j$
- $A_{i..j}$ can be obtained by splitting it into $A_{i..k}$ and $A_{k+1..j}$ and then multiplying the sub-products. $i \leq k < j$
- The cost of this parenthesization is:
  - cost of computing the matrix $A_{i..k}$ +
  - cost of computing the matrix $A_{k+1..j}$ +
  - cost of multiplying them together $(A_{i..k} \times A_{k+1..j}) = p_{i-1}p_kp_j$

Step 1: Optimal Sub-structure

- Consider $A_{1..4} = A_1 \times A_2 \times A_3 \times A_4$:
  - $A_{1..1}, A_{2..4}$ $\quad k = 1$
  - $A_{1..2}, A_{3..4}$ $\quad k = 2$
  - $A_{1..3}, A_{4..4}$ $\quad k = 3$
- There are j-i possible splits (i.e. $k = i, \ldots, j$-1)
  - In the above example, # of splits are 3 = 4-1

Step 1: Optimal Sub-structure

- The optimal substructure of this problem is as follows:
  - Suppose that an optimal parenthesization of $A_i A_{i+1} \ldots A_j$ splits the product between A_k and A_{k+1}.
- Within the optimal parenthesization of $A_{i..j}$,
  - the parenthesization of $A_{i..k}$ must be optimal
  - the parenthesization of $A_{k+1..j}$ must be optimal
- The same thing holds for the sub-chain: $A_{k+1} A_{k+2} \ldots A_j$

Step 1: Optimal Sub-structure

Minimal
Cost_$A_{1..6}$ + Cost_$A_{7..9}$+$p_0 p_6 p_9$

$A_1A_2A_3A_4A_5A_6A_7A_8A_9$

Suppose $((A_1A_2)(A_3((A_4A_5)A_6)))\ ((A_7A_8)A_9)$ is optimal

Then $(A_1A_2)\ (A_3((A_4A_5)A_6))$ must be optimal for $A_1A_2A_3A_4A_5A_6$

Otherwise, if $(A_1(A_2A_3))\ ((A_4A_5)A_6)$ is optimal for $A_1A_2A_3A_4A_5A_6$

Then $((A_1(A_2A_3))\ ((A_4A_5)A_6))\ ((A_7A_8)A_9)$ will be better than

$((A_1A_2)(A_3((A_4A_5)A_6)))\ ((A_7A_8)A_9)$

Step 2: Recursive Solution

- For MCM, sub-problems are:
  - problems of determining the min-cost of a parenthesization of $A_i A_{i+1} \ldots A_j$ for $1 <= i <= j <= n$
- Let $m[i, j]$ = min # of scalar multiplications needed to compute the matrix $A_{i \ldots j}$.
  - For the whole problem, we need to find $m[1, n]$
- Since $A_{i..j}$ can be obtained by breaking it into $A_{i..k}\ A_{k+1..j}$, We have (each A_i of size $p_{i-1} \times p_i$)
  - $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$

Step 2: Recursive Solution

- Thus

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j & \text{if } i < j \end{cases}$$

- The $m[i, j]$ values give the cost of optimal solutions to sub-problems.
- Let $s[i, j]$ be the value k where the optimal split occurs
  - $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$

Step 3: Computing the Optimal Costs

- A direct recursive implementation is exponential
  - there is a lot of duplicated work (why?)
  - each sub-problem may be computed many times in different branches of its recursion tree

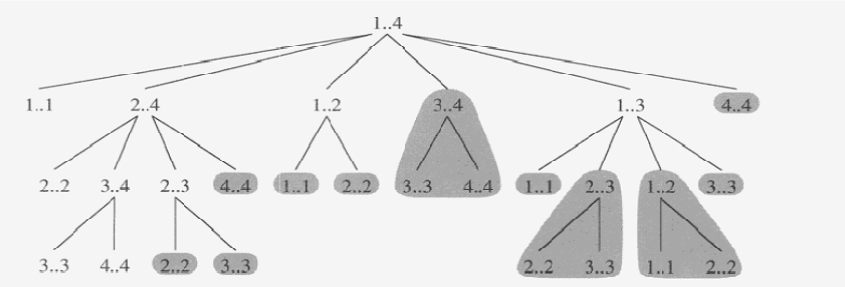


Figure 15.5 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN(p, 1, 4). Each node contains the parameters i and j. The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN(p, 1, 4).

Elements of Dynamic Programming

- DP is used to solve problems with the following characteristics:
  - Optimal substructure (Principle of Optimality)
    - optimal solution to the problem is constructed from the optimal solutions to subproblems
    - within the optimal parenthesization of $A_{i..j}$,
      - the parenthesization of $A_{i..k}$ must be optimal
      - the parenthesization of $A_{k+1..j}$ must be optimal
    - In MCM, $A_{1..6} = A_{1..3}\, A_{4..6}$
  - Overlapping subproblems
    - there exist some places where we solve the same subproblem **more than once**
    - In MCM, $A_{2..3}$ is common to the sub-problems $A_{1..3}$ and $A_{2..4}$
    - Effort wasted in solving common sub-problems repeatedly

Overlapping Sub-problems

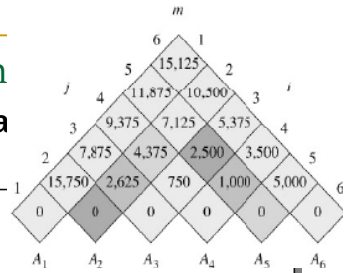
$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j\} & \text{if } i < j \end{cases}$$

```
Recursive-Matrix-Chain(p,i,j)

if i = j
   then return 0
m[i,j] = ∞
for k = i to j-1 do
   q = Recursive-Matrix-Chain(p,i,k)+
     Recursive-Matrix-Chain(p,k+1,j)+p_{i-1}p_k p_j
   if q < m[i,j]
      then m[i,j] = q
return m[i,j]
```


Step 3: Computing the Optim

The following code assumes that ma dimensions *pi*-1 x *pi* for *i* =1,2, …, *n*




```
Matrix-Chain-Order(⟨p_0, p_1, …, p_n⟩)

for i ← 1 to n do
  m[i,i] ← 0          // no multiplication for 1 matrix
for d ← 2 to n do          // d: length of sub-chain
  for i ← 1 to n-d+1 do    // i: start of sub-chain
    j ← i+d-1              // j: end of sub-chain
    m[i,j] ← ∞
    for k ← i to j-1 do
      q ← m[i,k] + m[k+1,j] + p_{i-1}p_k p_j
      if q < m[i,j] then
        m[i,j] ← q
        s[i,j] ← k
return m, s
```

Step 3: Computing the Optimal Costs

- $m[i, i] = 0$

```
Matrix-Chain-Order(⟨p0, p1, …, pn⟩)
2  for i ← 1 to n do
3    m[i,i] ← 0          // no multiplication for 1 matrix
```

- $m[i, i+1]$ for $i = 1, 2, \ldots, n-1$ for chains of length $d = 2$
- $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$ for chains of length $d = 3$
- …

```
4    for d ← 2 to n do          // d: length of sub-chain
5      for i ← 1 to n-d+1 do    // i: start of sub-chain
6        j ← i+d-1              // j: end of sub-chain
7        m[i,j] ← ∞
8        for k ← i to j-1 do
9          q ← m[i,k] + m[k+1,j] + p_{i-1}p_k p_j
10         if q < m[i,j] then
11           m[i,j] ← q
12           s[i,j] ← k
```


Step 3: Computing the Optimal Costs

- At each step, the $m[i, j]$ cost computed at lines 9-12 depends only on table entries $m[i, k]$ and $m[k+1, j]$ already computed.
- After execution:
  - $m[1, n]$ contains the value of the optimal solution and
  - s contains optimal subdivisions (choices of k) of any sub-problem into two sub-sub-problems
- Running time
  - It is easy to see that it is $O(n^3)$
    - $O(n^2)$ entries, $O(n)$ time to compute each ➔ $O(n^3)$ total time
  - It turns out, it is also $\Omega(n^3)$
  - From **exponential** time to **polynomial**

Step 3: Computing the Optimal Costs

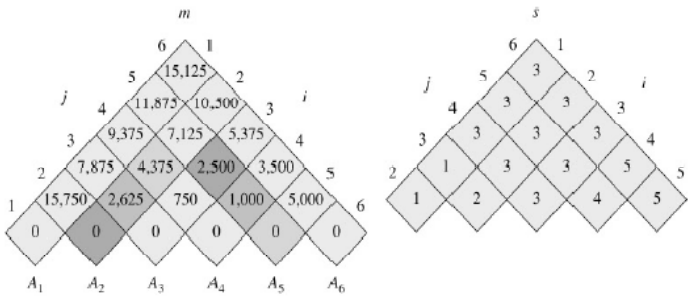
- **Example**:
- Solve the following MCM instance ($n = 6$):

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

- $p = \langle 30, 35, 15, 5, 10, 20, 25 \rangle$

Step 3: Computing the Optimal Costs

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25



$$ex: m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35.15.20 = 13000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35.5.20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35.10.20 = 11375 \end{cases}$$

$$= 7125.$$

Step 3: Computing the Optimal Costs

- The previous table shows the structure of the problem $m[1,6]$ and all the subproblems, $m[1,2]$, $m[1,3]$, etc.
- The problem $m[1,6]$ is at the top of the table with the cost of 15125.
- As an example, the subproblem $m[2,5]$ is highlighted in the darkly shaded square.
- Notice that this subproblem is made up of the further subproblems to the left and down.

Step 3: Computing the Optimal Costs

- The minimal cost of $m[2,5]$ is found by exhaustively searching all values of k.
  - $m[2,5] =$
  - $\min\{$
  - $m[2,2]+m[3,5]+p_1p_2p_5=0+2500+35*15*20 \quad = 13000,$
  - $m[2,3]+m[4,5]+p_1p_3p_5=2625+1000+35*5*20 = 7125,$
  - $m[2,4]+m[5,5]+p_1p_4p_5=4375+0+35*10*20 \quad = 11375\}$
  - $= 7125$

$$m[3,5] = \min\begin{cases} m[3,4]+m[5,5] + 15*10*20 \\ =750 + 0 + 3000 = 3750 \\ m[3,3]+m[4,5] + 15*5*20 \\ =0 + 1000 + 1500 = 2500 \end{cases}$$

Step 4: Constructing an Optimal Solution

- Although **Matrix-Chain-Order** determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it *does not* directly show how to multiply the matrices
- Using table ***s***, we can construct an optimal solution

Step 4: Constructing an Optimal Solution

- The previous table ***s*** table which is structured just like the ***m*** table.
- Each entry in the table is indexed by the $[i, j]$ pair.
- The ***s*** table contains the calculated ***k*** value for each entry in the ***m*** table.
- The ***s*** table can be used to formulate the optimal parenthesization of the matrix chain.
- This completes the structure of the problem

Step 4: Constructing an Optimal Solution

- Each $s[i, j]$ entry contains the calculated k value
  - such that the optimal parenthesization of $A_{i..j} = A_i A_{i+1} \ldots A_j$ splits the product between to
    - $A_{i..k} A_{k+1..j}$
- A simple recursive algorithm **Print-Optimal-Parents(*s*, *i*, *j*)** can be used to reconstruct an optimal parenthesization

Step 4: Constructing an Optimal Solution

- The initial call **Print-Optimal-Parents(*s*, 1, *n*)** prints an optimal parenthesization of $\langle A_1, A_2, \ldots, A_n \rangle$

```
Print-Optima-Parens(s, i, j)
if i = j
    then print "A"_i
    else print "("
         Print-Optimal-Parens(s, i, s[i, j])
         Print-Optimal-Parens(s, s[i, j]+1, j)
         Print ")"
```

- The optimal split of $A_{1..n}$ is stored in $s[1,n]$
  - $A_{1..s[1,n]} A_{s[1,n]+1..n}$

Step 4: Constructing an Optimal Solution

- To get the optimal solution $A_{1..6}$, s is used as follows:
  - $A_{1..6} = (A_{1..3}\, A_{4..6})$
    - since $s[1,6] = 3$
      $= ((A_{1..1}\, A_{2..3})(A_{4..5}\, A_{6..6}))$
    - since $s[1,3] = 1$ and $s[4,6] = 5$
      $= ((A_1(A_2\, A_3))((A_4\, A_5)A_6))$

```
Print-Optima-Parens(s, i, j)
if i = j
    then print "A"_i
    else print "("
         Print-Optimal-Parens(s, i, s[i, j])
         Print-Optimal-Parens(s, s[i, j]+1, j)
         Print ")"
```


Example: Matrix Chain Multiplication

- What if we want to multiply a string of more than two matrices together?
- Matrix multiplication is associative, that is A(BC) = (AB)C, so the result is the same no matter in what order we multiply the matrices, but it can make a big difference in the amount of work that is done.

A_1	x	A_2	x	A_3	x	A_4
30 x 1		1 x 40		40 x 10		10 x 25

$((A_1A_2)A_3)A_4$	30*1*40	+	30*40*10	+	30*10*25	= 20,700
$A_1(A_2(A_3A_4))$	40*10*25	+	1*40*25	+	30*1*25	= 11,750
$(A_1A_2)(A_3A_4)$	30*1*40	+	40*10*25	+	30*40*25	= 41,200
$A_1((A_2A_3)A_4)$	1*40*10	+	1*10*25	+	30*1*25	= 1,400

A_1 x A_2 x A_3 x A_4

30 x 1 1 x 40 40 x 10 10 x 25

[1]	[2]	[3]	[4]	
0				[1]
	0			[2]
		0		[3]
			0	[4]

First, the costs of multiplying chains of length 1 are initialized to 0

A_1 x A_2 x A_3 x A_4

30 x 1 1 x 40 40 x 10 10 x 25

A_1 x A_2
30*1*40

A_2 x A_3
1*40*10

A_3 x A_4
40*10*25

[1]	[2]	[3]	[4]	
0	1200			[1]
	0	400		[2]
		0	10000	[3]
			0	[4]

Then the costs of multiplying chains of length 2 are calculated
(only one way to multiply chains of length two)

A_1 x A_2 x A_3 x A_4

30 x 1 1 x 40 40 x 10 10 x 25

[1]	[2]	[3]	[4]	
0	1200	700		[1]
	0	400	650	[2]
		0	10000	[3]
			0	[4]

BEST

A_1 x (A_2 x A_3)
0 + 400 + 30*1*10 = 700

(A_1 x A_2) x A_3
1200 + 0 + 30*40*10 = 13,200

Then the best way to multiply chains of length 3 are calculated

A_1 x A_2 x A_3 x A_4

30 x 1 1 x 40 40 x 10 10 x 25

[1]	[2]	[3]	[4]	
0	1200	700		[1]
	0	400	650	[2]
		0	10000	[3]
			0	[4]

A_2 x (A_3 x A_4)
0 + 10,000 + 1*40*25 = 11,000

(A_2 x A_3) x A_4
400 + 0 + 1*10*25 = 650

BEST

A_1 x A_2 x A_3

30 x 1 1 x 40 40 x

Then the best way to Multiply the entire chain is calculated

[1]	[2]	[3]	[4]	
0	1200	700	1400	[1]
	0	400	650	[2]
		0	10000	[3]
			0	[4]

BEST

A_1 x (A_2 x A_3 x A_4)
0 + 650 + 30*1*25 = 1400

(A_1 x A_2 x A_3) x A_4
700 + 0 + 30*10*25 = 10,450

(A_1 x A_2)(A_3 x A_4)
1200 + 10,000 + 30*40*25 = 41,200

A_1 x A_2 x A_3 x A_4

30 x 1 1 x 40 40 x 10 10 x 25

cost =

[1]	[2]	[3]	[4]	
0	1200	700	1400	[1]
	0	400	650	[2]
		0	10000	[3]
			0	[4]

split =

	[2]	[3]	[4]	
	1	1	1	[1]
		2	3	[2]
			3	[3]

the array split is used to reconstruct the optimal ordering

Dynamic Programming Solution

```
PRINT-OPTIMAL-PARENS(split, i, j)
  if(i == j)    // base case -- chain of length 1, just print A_i
      print "A"_i
  else
      print "("
      PRINT-OPTIMAL-PARENS(split, i, split[i,j])
      PRINT-OPTIMAL-PARENS(split, split[i,j]+1, j)
      print ")"
```


	[2]	[3]	[4]	
	1	1	1	[1]
		2	3	[2]
			3	[3]

P-O-P(split,1,4)

P-O-P(split,1,1)

P-O-P(split,2,4)

P-O-P(split,2,3)

P-O-P(split,4,4)

P-O-P(split,2,2)

P-O-P(split,3,3)

(A_1 ((A_2 A_3) A_4))

Longest Common Subsequence

- In biological applications, we often want to compare the DNA of two (or more) different organisms.
- A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, quanine, cytosine, and thymine (A, C, G, T).

Longest Common Subsequence

Similarity can be defined in different ways:

- Two DNA strands are similar if one is a substring of the other.
- Two strands are similar if the number of changes needed to turn one into the other is small.
- There is a thirds strand S_3 in which the bases in S_3 appear in each of S_1 and S_2; these bases must appear in the same order, but not necessarily consecutively. The longer the strand S3 we can find, the more similar S1 and S2 are.

Example of Longest Common Subsequence

LCS for S_1 and S_2:

S_1=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
S_2=GTCGTTCGGAATGCCGTTGCTCTGTAAA
S_3=GTCGTCGGAAGCCGCCGAA

Example

X = ⟨A, B, C, B, D, A, B⟩
Y = ⟨B, D, C, A, B, A⟩

X = ⟨A, B, C, B, D, A, B⟩
Y = ⟨B, D, C, A, B, A⟩

- ⟨B, C, B, A⟩ and ⟨B, D, A, B⟩ are longest common subsequences of X and Y (length = 4)
- ⟨B, C, A⟩, however is not a LCS of X and Y

LCS Brute-force solution

- Enumerate all subsequences of ***X*** and check each subsequence to see if it is also a subsequence of ***Y***, keeping track of the longest subsequence found.
- Each subsequence of ***X*** corresponds to a subset of the indices {1, 2, …, *m*} of ***X***.
- There are 2^m subsequences of ***X***.
- Exponential Time!

A Recursive Solution

Case 1: $x_i = y_j$

e.g.: X = ⟨A, B, D, E⟩

Y = ⟨Z, B, E⟩

$$c[i, j] = c[i - 1, j - 1] + 1$$

- Append $x_i = y_j$ to the LCS of X_{i-1} and Y_{j-1}
- Must find a LCS of X_{i-1} and Y_{j-1} ⇒ optimal solution to a problem includes optimal solutions to sub problems

A Recursive Solution

Case 2: $x_i \neq y_j$

e.g.: X = ⟨A, B, D, G⟩

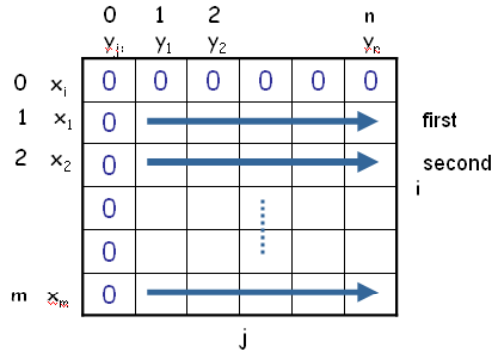
Y = ⟨Z, B, D⟩

$$c[i, j] = \max\{ c[i - 1, j], c[i, j-1] \}$$

- Must solve two problems
  - find a LCS of X_{i-1} and Y_j: X_{i-1} = ⟨A, B, D⟩ and Y_j = ⟨Z, B, D⟩
  - find a LCS of X_i and Y_{j-1}: X_i = ⟨A, B, D, G⟩ and Y_j = ⟨Z, B⟩
- Optimal solution to a problem includes optimal solutions to subproblems

Computing the Length of the LCS

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$



Additional Information

$$c[i, j] = \begin{cases} 0 & \text{if } i, j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$

b & c:

		0 y_j	1 A	2 C	3 D		n F
0	x_i	0	0	0	0	0	0
1	A	0					
2	B	0			c[i-1,j]		
3	C	0		↖ c[i,j-1]	↑		
		0					
m	D	0					

i

j

A matrix b[i, j]:

- For a subproblem [i, j] it tells us what choice was made to obtain the optimal value
- If $x_i = y_j$
  b[i, j] = "↖"
- Else, if
  c[i - 1, j] ≥ c[i, j-1]
  b[i, j] = "↑"
  else
  b[i, j] = "←"

```
LCS-Length(X, Y)   // dynamic programming solution
   m = X.length()
   n = Y.length()
   for i = 0 to m do  c[i,0] = 0
   for j = 0 to n do  c[0,j] = 0                    O(nm)
   for i = 1 to m do  // row
        for j = 1 to n do // cloumn
              if x_i = =y_i then
                    c[i,j] = c[i-1,j-1] + 1
                    b[i,j] =“↖”
              else if c[i-1, j] ≥ c[i,j-1] then
                    c[i,j] = c[i-1,j]
                    b[i,j] = “^”
              else  c[i,j] = c[i,j-1]
                    b[i,j] = “<”
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

First Optimal-LCS initializes row 0 and column 0

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	^ 0	^ 0	^ 0	↖ 1	< 1	1
2	B	0	↖ 1	< 1	< 1	^ 1	↖ 2	< 2
3	C	0	^ 1	^ 1	↖ 2			
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Next each c[*i*, *j*] is computed, row by row, starting at c[1,1].
If $x_i == y_j$ then c[i, j] = c[i-1, j-1]+1 and b[i, j] = ↖

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	^0	^0	^0	↖1	< 1	1
2	B	0	↖1	< 1	< 1	^1	↖2	< 2
3	C	0	^1	^1	↖2	< 2		
4	B	0						
5	D	0						
6	A	0						
7	B	0						

If x_i <> y_j then c[i, j] = max(c[i-1, j], c[i, j-1]) and b[i, j] points to the larger value

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	^0	^0	^0	↖1	< 1	1
2	B	0	↖1	< 1	< 1	^1	↖2	< 2
3	C	0	^1	^1	↖2	< 2	^2	
4	B	0						
5	D	0						
6	A	0						
7	B	0						

if c[i-1, j] == c[i, j-1] then b[i,j] points up

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	^0	^0	^0	↖1	< 1	↖1
2	B	0	↖1	< 1	< 1	^1	↖2	< 2
3	C	0	^1	^1	↖2	< 2	^2	^2
4	B	0	↖1	^1	^2	^2	↖3	< 3
5	D	0	^1	↖2	^2	^2	^3	^3
6	A	0	^1	^2	^2	↖3	^3	↖4
7	B	0	↖1	^2	^2	^3	↖4	^4

To construct the LCS, start in the bottom right-hand corner and follow the arrows. A ↖ indicates a matching character.

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	^0	^0	^0	↖1	< 1	↖1
2	B	0	↖1	< 1	< 1	^1	↖2	< 2
3	C	0	^1	^1	↖2	< 2	^2	^2
4	B	0	↖1	^1	^2	^2	↖3	< 3
5	D	0	^1	↖2	^2	^2	^3	^3
6	A	0	^1	^2	^2	↖3	^3	↖4
7	B	0	↖1	^2	^2	^3	↖4	^4

LCS: B C B A

Constructing an LCS

```
Print-LCS(b,X,i,j)
if i = 0 or j = 0 then
   return
   if b[i,j] = "↖" then
      Print-LCS(b, X, i-1, j-1)
        print x_i
   else if b[i,j] = "^" then
      Print-LCS(b, X, i-1, j)
   else Print-LCS(b, X, i, j-1)
```

Knapsack problem

- There are two versions of the problem:

(1) "0-1 knapsack problem" and

(2) "Fractional knapsack problem"

(1) Items are indivisible; you either take an item or not. Solved with *dynamic programming*

(2) Items are divisible: you can take any fraction of an item. Solved with a *greedy algorithm*.

0-1 Knapsack problem

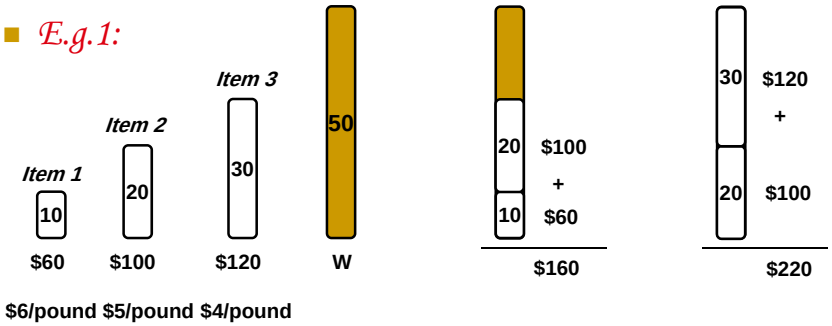
- Thief has a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value v_i (all w_i, v_i and W are integer values)
- <u>Problem</u>: How to pack the knapsack to achieve maximum total value of packed items?
- Goal:
  - find x_i such that for all $x_i = \{0, 1\}$, $i = 1, 2, .., n$

    $\sum w_i x_i \leq W$ and

    $\sum x_i v_i$ is maximum

0-1 Knapsack - Greedy Strategy

- *E.g.1:*



- *E.g.2:*
  - **Item: 1 2 3 4 5 6 7**
  - **Benefit: 5 8 3 2 7 9 4**
  - **Weight: 7 8 4 10 4 6 4**
  - **Knapsack holds a maximum of 22 pounds**
  - **Fill it to get the maximum benefit**

0-1 Knapsack problem: brute-force approach

- Let's first solve this problem with a straightforward algorithm
- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to W
- Running time will be $O(2^n)$

0-1 Knapsack - Dynamic Programming

- P(i, w) – the maximum profit that can be obtained from items 1 to i, if the knapsack has size w
- Case 1: thief takes item i

  $P(i, w) = v_i + P(i - 1, w - w_i)$
- Case 2: thief does not take item i

  $P(i, w) = P(i - 1, w)$

Recursive Formula

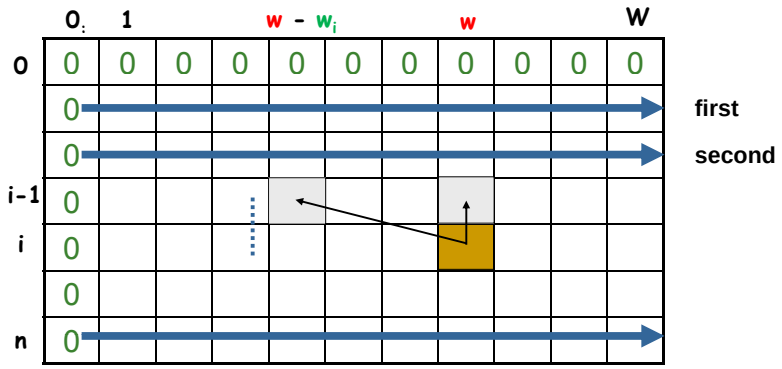
$$P[i,w] = \begin{cases} P[i-1,w] & \text{if } w_i > w \\ \max\{v_i + P[i-1, w-w_k], P[i-1,w]\} & \text{else} \end{cases}$$

- The best subset that has the total weight w, either contains item i or not.
- First case: $w_i > w$. Item i can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable
- Second case: $w_i <= w$. Then the item i <u>can</u> be in the solution, and we choose the case with greater value.

0-1 Knapsack - Dynamic Programming

Item i was taken | Item i was not taken

$$P(i, w) = \max\{v_i + P(i - 1, w - w_i), P(i - 1, w)\}$$



Example:

$P(i, w) = \max\{v_i + P(i-1, w-w_i), P(i-1, w)\}$

W = 5

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

P(1, 1) = P(0, 1) = 0

P(1, 2) = max{12+0, 0} = 12

P(1, 3) = max{12+0, 0} = 12

P(1, 4) = max{12+0, 0} = 12

P(1, 5) = max{12+0, 0} = 12

P(2, 1)= max{10+0, 0} = 10

P(2, 2)= max{10+0, 12} = 12

P(2, 3)= max{10+12, 12} = 22

P(2, 4)= max{10+12, 12} = 22

P(2, 5)= max{10+12, 12} = 22

P(3, 1)= P(2,1) = 10

P(3, 2)= P(2,2) = 12

P(3, 3)= max{20+0, 22}=22

P(3, 4)= max{20+10,22}=30

P(4, 5)= max{20+12,22}=32

P(4, 1)= P(3,1) = 10

P(4, 2)= max{15+0, 12} = 15

P(4, 3)= max{15+10, 22}=25

P(4, 4)= max{15+12, 30}=30

P(4, 5)= max{15+22, 32}=37

Reconstructing the Optimal Solution

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

- Item 4
- Item 2
- Item 1

- Start at P(n, W)
- When you go left-up ⇒ item i has been taken
- When you go straight up ⇒ item i has not been taken

Optimal Substructure

- Consider the most valuable load that weights at most W pounds
- If we remove item j from this load

⇒ The remaining load must be the most valuable load weighing at most $W - w_j$ that can be taken from the remaining n - 1 items

0-1 Knapsack Algorithm

```
for w = 0 to W
   P[0,w] = 0
for i = 0 to n
   P[i,0] = 0
   for w = 0 to W
      if w_i <= w // item i can be part of the solution
         if v_i + P[i-1,w-w_i] > P[i-1,w]
            P[i,w] = v_i + P[i-1,w- w_i]
         else
            P[i,w] = P[i-1,w]
      else P[i,w] = P[i-1,w]  // w_i > w
```

O(n*W)

Overlapping Subproblems

$P(i, w) = \max \{v_i + P(i - 1, w-w_i), P(i - 1, w) \}$

	0	1				w					W
0	0	0	0	0	0	0	0	0	0	0	0
	0										
	0										
i-1	0										
i	0										
	0										
n	0										

E.g.: all the subproblems shown in grey may depend on P(i-1, w)

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary