# Dynamic programming

# Outline

- What is dynamic programming ?
- Main steps in applying dyamic programming
- Recurrence relations: ascending versus descendent
- Applications

# What is dynamic programming ?

- An algorithm design technique for solving problems that can be decomposed in overlapping subproblems – can be applied to optimization problems with optimal substructure property.

- Main feature: each suproblem solved once. Its solution is stored in a table and then used to solve initial problem.

Obs.

- Developed in the 1950's by Richard Bellman as a general optimization method.
- "Programming" in DP refers to planning and not coding on a computer.
- Dynamic = manner that one constructs tables holding partial solutions

# Main steps in applying dynamic programming

Analyze problem structure: establish the way in which the solution of the problem depends on solutions of subproblems .

Identify/develop recurrence relation connecting problem and subproblem solutions. Usually the recurrence relation involves the optimum criterion.

Developing solution

# Outline

- What is dynamic programming ?
- Main steps in applying dynamic programming
- Recurrence relations: ascending versus descendent
- Applications

# Recurrence relations

Two main approaches:

- bottom up: start from base case and generate new values.
- top down: value to compute is expressed by previous values that have to be,in turn, computed. Usually implemented recursively, inefficient unless we use memoization.

# Recurrence relations

Exemplu 1. m-th element of Fibonacci seq.

$f_1=f_2=1$; $f_n=f_{n-1}+f_{n-2}$ for $n>2$

Top down (recursive):

```
fib(m)
IF (m=1) OR (m=2)  THEN
    RETURN 1
ELSE
   RETURN fib(m-1)+fib(m-2)
ENDIF
```

Complexity:

$$T(m) = \begin{cases} 0 & \text{if } m<=2 \\ T(m-1)+T(m-2)+1 & \text{if } m>2 \end{cases}$$

T:

0 0 1 2 4 7 12 20 33 54 …

Fibonacci:

1 1 2 3 5 8 13 21 34 55 …

$f_n = O(phi^n)$,

phi=(1+sqrt(5))/2

exponential complexity!

# Recurrence relation

Exemplu 1. m-lea element of Fibonacci sequence

$f_1=f_2=1$; $f_n=f_{n-1}+f_{n-2}$ for n>2

Bottom up:

```
fib(m)
f[1]←1; f[2] ← 1;
FOR i ← 3,m DO
    f[i] ← f[i-1]+f[i-2]
ENDFOR
RETURN f[m]
```

Complexity:

T(m)=m-2 => linear complexity

Obs: time efficienccy compensated by using more space

```
fib(m)
f1 ← 1; f2 ← 1;
FOR i ← 3,m DO
    f2 ← f1+f2; f1 ← f2-f1;
ENDFOR
RETURN f2
```

# Recurrence relations

Example 2. binomial coefficients C(n,k) (combinari de n luate cate k)

$$C(n,k)= \begin{cases} 0 & \text{if } n<k \\ 1 & \text{if } k=0 \text{ or } n=k \\ C(n-1,k)+C(n-1,k-1) & \text{otherwise} \end{cases}$$

Top down:

```
comb(n,k)
  IF (k=0) OR (n=k) THEN
     RETURN 1
  ELSE
     RETURN comb(n-1,k)+comb(n-1,k-1)
  ENDIF
```

Complexity:

Dim pb: (n,k)

Dominant op: addition

$$T(n,k)= \begin{cases} 0 & \text{if } k=0 \text{ or } k=n \\ T(n-1,k)+T(n-1,k-1) & \end{cases}$$

Nr additions = nr nodes in recursive call tree.

$T(n,k) >= 2^{\min\{k,n-k\}}$

$T(n,k) \in \Omega(2^{\min\{k,n-k\}})$

# Recurrence relations

Exemplu 2. Computing binomial coefficients C(n,k)

$$C(n,k)=\begin{cases} 0 & \text{if } n<k \\ 1 & \text{if } k=0 \text{ sau } n=k \\ C(n-1,k)+C(n-1,k-1) & \text{otherwise} \end{cases}$$

Bottom up: Pascal's triangle

| | 0 | 1 | 2 | … | k-1 | k |
|---|---|---|---|---|---|---|
| 0 | 1 | | | | | |
| 1 | 1 | 1 | | | | |
| 2 | 1 | 2 | 1 | | | |
| … | | | | | | |
| k | 1 | | | … | | 1 |
| … | | | | | | |
| n-1 | 1 | | | | C(n-1,k-1) | C(n-1,k) |
| n | 1 | | | | | C(n,k) |

# Recurrence relations

**Algorithm:**

```
Comb(n,k)
FOR i←0,n DO
  FOR j ← 0,min{i,k} DO
    IF (j=0) OR (j=i) THEN
     C[i,j] ← 1
    ELSE
     C[i,j] ← C[i-1,j]+C[i-1,j-1]
    ENDIF
  ENDFOR
ENDFOR
RETURN C[n,k]
```

**Complexity:**

Dimension of the pb: (n,k)

Dominant operation: addition

$$T(n,k)=(1+2+\ldots+k-1)+(k+\ldots+k)$$
$$=k(k-1)/2+k(n-k+1)$$

$$T(n,k) \in \Theta(nk)$$

Obs. If we only have to compute C(n,k) it is enough to use a table with k elements as a additional space.

# Outline

- What is dynamic programming ?
- Main steps in applying dynamic programming
- Recurrence relations: ascending versus descendent
- Applications

# Applications of dynamic programming

## Longest (strictly) increasing sequence

Let $a_1,a_2,\ldots,a_n$ be a sequence. Find the longest subsequence such that

$a_{j1}<a_{j2}<\ldots<a_{jk}$

Example:

a = (2,5,1,3,6,8,2,10,4)

Increasing subsequences of length 5 (maximum length):

(2,5,6,8,10)

(2,3,6,8,10)

(1,3,6,8,10)

# Longest increasing subsequene

1. Analysis.

Let s=($a_{j1}$, $a_{j2}$,…,$a_{j(k-1)}$ ,$a_{jk}$ ) be the optimal solution. Then none of the elements in a[1..n] after $a_{jk}$ is greater than $a_{jk}$. Moreover, no element with index between $j_{(k-1)}$ and $j_k$ has a value between corresponding elements of subsequence s (or s would no longer be optimal).

Show that s'=($a_{j1}$, $a_{j2}$,…,$a_{j(k-1)}$ ) is an optimal soln for LIS ending in $a_{j(k-1)}$ . Assume s' not optimal. Then there is a longer subsequence s". Adding to s" element $a_{jk}$ we obtain a solution better than s, contradicting the fact that s is optimal.

In conclusion: the problem has the optimal substructure property.

# Longest increasing subsequence

1. Building a recurrence relation

Let $B_i$ be the number of elements of a LIS ending in $a_i$

$$B_i = \begin{cases} 1 & \text{if } i=1 \\ 1+ \max\{B_j \mid 1<=j<=i-1, a_j<a_i\} \end{cases}$$

Exemplu:

a = (2,5,1,3,6,8,2,10,4)

B = (1,2,1,2,3,4,2,5,3)

# Longest increasing subsequence

3. Recurrence relation

$$B_i = \begin{cases} 1 & \text{if } i=1 \\ 1+\max\{B_j \mid 1<=j<=i-1,\ a_j<a_i\} \end{cases}$$

Complexity: $\theta(n^2)$

```
calculB(a[1..n])
B[1]←1
FOR i ← 2,n DO
  max ← 0
  FOR j ← 1,i-1 DO
    IF a[j]<a[i] AND max<B[j]
       THEN max ← B[j]
    ENDIF
  ENDFOR
  B[i] ← max+1
ENDFOR
RETURN B[1..n]
```

# Longest increasing subsequene

1. Constructing the solution

Determine the maximum of B

Construct s succesively starting from last element

Complexity: θ(n)

```
construire(a[1..n],B[1..n])
m ← 1
FOR i ← 2,n DO
   IF B[i]>B[m] THEN m ← i ENDIF
ENDFOR
k ← B[m]
s[k] ← a[m]
WHILE B[m]>1 DO
    i ← m-1
   WHILE a[i]>=a[m] OR B[i]<>B[m]-1 DO
        i ← i-1
   ENDWHILE
   m ← i;  k ← k-1;  s[k] ← a[m]
ENDWHILE
RETURN s[1..k]
```

# Longest increasing subsequence

```
calculB(a[1..n])
B[1]:=1;  P[1]:=0
FOR i:=2,n DO
   max:=0
   P[i]:=0
   FOR j:=1,i-1 DO
      IF a[j]<a[i] AND max<B[j]
      THEN max:=B[j]
               P[i]:=j
      ENDIF
   ENDFOR
   B[i]:=max+1
ENDFOR
RETURN B[1..n]
```

```
construire(a[1..n],B[1..n],P[1..n])
m:=1
FOR i:=2,n DO
   IF B[i]>B[m] THEN m:=i ENDIF
ENDFOR
k:=B[m]
s[k]:=a[m]
WHILE P[m]>0 DO
     m:=P[m]
     k:=k-1
     s[k]:=a[m]
ENDWHILE
RETURN s[1..k]
```

P[i] is the index of the element preceding a[i] in optimal subsequence.
Using P[1..n] simplifies constructing the solution

# Longest common subsequence

Example:

a: 2 1 4 3 2

b: 1 3 4 2

Common subsequences:

1, 3

1, 2

4, 2

1, 3, 2

1, 4, 2

Variant: determine LCS consisting of consecutive elements

Example:

a: 2 1 3 4 5

b: 1 3 4 2

Common subsequences:

1, 3

3, 4

1, 3, 4

# Longest common subsequence

1. Structure of optimal solutions

Let P(i,j) be the problem determining LCS of sequences a[1..i] and b[1..j]. If a[i]=b[j] then the optimal solution contains this common element; the rest is represented by optimal solution of P(i-1,j-1) (i.e. determining LCS of a[1..i-1] and b[1..j-1]). If a[i]<>b[j] then optimal solution coincides to the best of the solutions of subproblems P(i-1,j) and P(i,j-1).

1. Recurrence relation. Let L(i,j) the length of the optimal solution of P(i,j). Then:

$$L[i,j]=\begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1+L[i-1,j-1] & \text{if } a[i]=b[j] \\ \max\{L[i-1,j],L[i,j-1]\} & \text{othereise} \end{cases}$$

# Longest common subsequence

Example:

a: 2 1 4 3 2

b: 1 3 4 2

$$L[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0\\ 1+L[i-1,j-1] & \text{if } a[i]=b[j]\\ \max\{L[i-1,j],L[i,j-1]\} & \text{otherwise}\end{cases}$$

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 1 | 1 | 2 | 2 |
| 4 | 0 | 1 | 2 | 2 | 2 |
| 5 | 0 | 1 | 2 | 2 | 3 |

# Longest common sequence

Recurrence relation:

$$L[i,j]=\begin{cases} 0 & \text{daca} \quad i=0 \quad \text{sau} \quad j=0 \\ 1+L[i-1,j-1] & \text{daca } a[i]=b[j] \\ \max\{L[i-1,j],L[i,j-1]\} & \text{altfel} \end{cases}$$

```
calcul(a[1..n],b[1..m])
FOR i:=0,n DO L[i,0]:=0 ENDFOR
FOR j:=1,m DO L[0,j]:=0 ENDFOR
FOR i:=1,n DO
  FOR j:=1,m DO
   IF a[i]=b[j]
   THEN  L[i,j]:=L[i-1,j-1]+1
   ELSE
     L[i,j]:=max(L[i-1,j],L[i,j-1])
   ENDIF
ENDFOR ENDFOR
RETURN L[0..n,0..m]
```

# Longest increasing subsequence

Constructing the solution (recursively):

```
Construction(i,j)
IF i>=1 AND j>=1 THEN
   IF a[i]=b[j]
  THEN
      construction(I-1,j-1)
      k:=k+1
      c[k]:=a[I]
  ELSE
      IF L[i-1,j]>L[i,j-1]
      THEN  construction(i-1,j)
      ELSE  construction(i,j-1)
ENDIF ENDIF ENDIF
```

Observations:

- a, b, c si k are global vars
- Before calling the function we initialize k (k:=0)
- Main call:

  construction(n,m)

# Application: discrete knapsack

The knapsack problem

Let us consider a set of n objects. Each object is characterized by its weight (or dimension - d) and its value (or profit - p). We want to fill in a knapsack of capacity C such that the total value of the selected objects is maximal.

Variants:

(i) Continuous variant: entire objects or part of objects can be selected. The components of the solution are from [0,1].

(ii) Discrete variant (0-1): an object either is entirely transferred into the knapsack or is not transferred. The solution components are from {0,1}

# Application: discrete knapsack

Assumption:

the capacity C and the dimensions $d_1,\ldots,d_n$ are natural numbers

The problem can be reformulated as:

find $(s_1,s_2,\ldots,s_n)$ with $s_i$ in $\{0,1\}$ such that:

$s_1 d_1 + \ldots + s_n d_n \le C$ (constraint)

$s_1 p_1 + \ldots + s_n p_n$ is maximal (optimization criterion)

Remark

the greedy technique can be applied but it does not guarantee the optimality

# Application: discrete knapsack

Example: n=3,

C=5,

d1=1, d2=2, d3=3

p1=6, p2=10, p3=12

Relative profit:

pr1=6, pr2=5, pr3=4

Greedy idea:

- Sort decreasingly the set of objects on the relative profit ($p_i/d_i$)
- Select the elements until the knapsack is filled

Greedy solution: (1,1,0)

Total value: V=16

Remark: this is not the optimal solution;

the solution (0,1,1) is better since V=22

Assumption: the object sizes and the knapsack capacity are natural numbers

# Application: discrete knapsack

1. Analyzing the structure of an optimal solution

Let P(i,j) be the generic problem of selecting from the set of objects $\{o_1,\ldots,o_i\}$ in order to fill in a knapsack of capacity j.

Remarks:

- P(n,C) is the initial problem
- If i<n, j<C then P(i,j) is a subproblem of P(n,C)
- Let s(i,j) be an optimal solution of P(i,j). There are two situations:
  - $s_i=1$ (the object $o_i$ is selected) => this lead us to the subproblem $P(i-1,j-d_i)$ and if s(i,j) is optimal then $s(i-1,j-d_i)$ should be optimal
  - $s_i=0$ (the object $o_i$ is not selected) => this lead us to the subproblem P(i-1,j) and if s(i,j) is optimal then s(i-1,j) should be optimal

Thus the solution s has the optimal substructure property

# Application: discrete knapsack

2. Find a recurrence relation

Let $V(i,j)$ be the total value corresponding to an optimal solution of $P(i,j)$

$$V(i,j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ V(i-1,j) & \text{if } d_i>j \text{ or } V(i-1,j)>V(i-1,j-d_i)+p_i \\ V(i-1,j-d_i)+p_i & \text{otherwise} \end{cases}$$

(the set is empty or the knapsack has not capacity at all)

(either the object i doesn't fit the knapsack or by selecting it we obtain a worse solution than by not selecting it)

# Application: discrete knapsack

The recurrence relation can be written also as:

$$V(i,j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ V(i-1,j) & \text{if } d_i>j \\ \max\{V(i-1,j),\ V(i-1,j-d_i)+p_i\} & \text{if } d_i<=j \end{cases}$$

Remarks:

- for the problem P(n,C) the table V has (n+1) rows and (C+1) columns
- V(n,C) gives us the value corresponding to the optimal solution

# Application: discrete knapsack

Example:

$$V(i,j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ V(i-1,j) & \text{if } d_i>j \\ \max\{V(i-1,j), \\ \quad V(i-1,j-d_i)+p_i\} & \text{if } d_i<=j \end{cases}$$

d: 1 2 3

p: 6 10 12

V

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 6 | 6 | 6 | 6 | 6 |
| 2 | 0 | 6 | 10 | 16 | 16 | 16 |
| 3 | 0 | 6 | 10 | 16 | 18 | 22 |

# Application: discrete knapsack

3. Developing the recurrence relation

$$V(i,j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ V(i-1,j) & \text{if } d_i>j \\ \max\{V(i-1,j),\ V(i-1,j-d_i)+p_i\} & \text{if } d_i<=j \end{cases}$$

Algorithm:

```
computeV (p[1..n],d[1..n],C)
  FOR i:=0,n DO V[i,0]:=0 ENDFOR
  FOR j:=1,n DO V[0,j]:=0 ENDFOR
  FOR i:=1,n DO
     FOR j:=1,C  DO
       IF j<d[i] THEN V[i,j]:=V[i-1,j]
       ELSE
          V[i,j]:=max(V[i-1,j],V[i-1,j-d[i]]+p[i])
       ENDIF
     ENDFOR
  ENDFOR
  RETURN V[0..n,0..C]
```

# Application: discrete knapsack

1. Constructing the solution

Example:

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 6 | 6 | 6 | 6 | 6 |
| 2 | 0 | 6 | 10 | 16 | 16 | 16 |
| 3 | 0 | 6 | 10 | 16 | 18 | 22 |

Steps:

- Compare V[3,5] with V[2,5]. Since they are different it means that the object o3 is selected
- Go to V[2,5-$d_3$]=V[2,2]=10 and compare it with V[1,2]=6. Since they are different it means that also $o_2$ is selected
- Go to V[1,2-$d_2$]=V[1,0]=0. Since the current capacity is 0 we cannot select another object

Thus the solution is {$o_2$,$o_3$} or s=(0,1,1)

# Application: discrete knapsack

1. Constructing the solution

Example:

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 6 | 6 | 6 | 6 | 6 |
| 2 | 0 | 6 | 10 | 16 | 16 | 16 |
| 3 | 0 | 6 | 10 | 16 | 18 | 22 |

Algorithm:

```
Construct(V[0..n,0..C],d[1..n])
  FOR i:=1,n DO s[i]:=0 ENDFOR
  i:=n;   j:=C
  WHILE i>0 and j>0 DO
     WHILE (i>1) AND (V[i,j]=V[i-1,j])
           DO i:=i-1
     ENDWHILE
     s[i]:=1
     j:=j-d[i]
     i:=i-1
  ENDWHILE
  RETURN s[1..n]
```

# Application: discrete knapsack

Remark

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 6 | 6 | 6 | 6 | 6 |
| 2 | 0 | 6 | 10 | 16 | 16 | 16 |
| 3 | 0 | 6 | 10 | 16 | 18 | 22 |

To compute V[3,5] and to construct the solution only the marked values are needed

Thus the number of computations could be reduced by computing only the values which are necessary

We can do this by combining the top-down approach with the idea of storing the computed values in a table

This is the so-called memoization technique

# Memory functions (memoization)

Goal: solve only the subproblems that are necessary and solve them only once

Basic idea: combine the top-down approach with the bottom-up approach

Motivation:

- The classic top-down approach solves only the necessary subproblems but common subproblems are solved more than once (this leads to an inefficient algorithm)

- The classic bottom-up approach solves all subproblems but even the common ones are solved only once

# Memory functions (memoization)

## Steps in applying the memoization:

- Initialize the table with a virtual value (this value should be different from any value which could be obtained during the computations)
- Compute the value we are searching for (e.g. V[n,C]) in a recursive manner by storing in the same time the computed values in the table and using these values any time it is possible

Remark: p[1..n], d[1..n] and V[0..n,0..C] are global variables
Call: comp(n,C)

## Virtual initialization:

```
FOR i:=0,n DO
  FOR j:=0,C DO  V[i,j]:=-1 ENDFOR
ENDFOR
```

## Recursive function:

```
comp(i,j)
IF V[i,j]<>-1 THEN RETURN V[i,j]
ELSE
  IF i=0 OR j=0 THEN V[i,j]:=0
  ELSE
    IF j<d[i] THEN V[i,j]:=comp(i-1,j)
    ELSE
      V[i,j] :=
          max(comp(i-1,j),comp(i-1,j-d[i])+p[i])
    ENDIF  ENDIF
  RETURN V[i,j]
ENDIF
```

# Application: optimal multiplication of matrices

Given n matrices $A_1, A_2, \ldots, A_n$ to be multiplied in this order determine how to group the matrices such that the number of scalar multiplications is minimized

Remarks

1. The dimensions of matrices are compatible. Let us suppose that they are denoted by $p_0,p_1,\ldots p_n$ and the matrix $A_i$ has $p_{i-1}$ rows and $p_i$ columns

1. Different groupings of factors lead to the same result (since matrices multiplication is associative) but they can lead to different values for the number of scalar multiplications

# Application: optimal multiplication of matrices

Example: Let $A_1$, $A_2$ and $A_3$ be three matrices having the dimensions: (2,20), (20,5) and (5,10)

$p_0=2 \quad p_1=20 \quad p_2=5 \quad p_3=10$

We consider the following groupings:

- $(A_1*A_2)*A_3$ - this needs (2*20*5)+2*5*10=300 scalar multiplications

- $A_1*(A_2*A_3)$ – this needs (20*5*10)+2*20*10=1400 scalar multiplications

Remark: for large values of n the number of possible groupings can be very large

# Application: optimal multiplication of matrices

In the general case the grouping process is a hierarchical one:

- The upper level define the grouping corresponding to the last multiplication
- The other levels correspond to groupings of the remaining factors

We identify a grouping by the position of the last multiplication. For instance the grouping

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

is specified by the value k

There are (n-1) possible groupings at the upper level (1<=k<n-1) but to each upper level grouping correspond a lot of groupings of the two factors $A_1*\ldots*A_k$ and $A_{k+1}*\ldots*A_n$

# Application: optimal multiplication of matrices

The numbers of groupings for a product of n factors is:

$$K(n)=\begin{cases} 1 & n<=2 \\ K(1)*K(n-1)+\ldots+K(i)*K(n-i)+\ldots+K(n-1)*K(1) & n>2 \end{cases}$$

Remark:

K(n)=C(n-1) where C(0),C(1) … are the Catalan's numbers which satisfy:

$$C(n)=Comb(2n, n)/(n+1)$$

The order of K(n) is almost $4^{n-1}/(n-1)^{3/2}$

Thus an exhaustive search is not at all efficient !

# Application: optimal multiplication of matrices

1. Analyzing the structure of an optimal solution

Let us denote by A(i..j) the product $A_i*A_{i+1}*\dots*A_j$ (i<=j)

If the optimal multiplication corresponds to a grouping at position k (i<=k<j) then the computation of A(i..k) and A(k+1..j) should also be optimal (otherwise the computation of A(i..j) wouldn't be optimal)

Thus the property of optimal substructure is satisfied

# Application: optimal multiplication of matrices

## 2. Constructing a recurrence relation

Let us denote by c(i,j) the number of scalar multiplications necessary to compute A(i..j).

$$c(i,j)=\begin{cases} 0 & \text{if } i=j \\ \min\{c(i,k)+c(k+1,j)+p_{i-1}p_kp_j \mid i<=k<j\} & \text{if } i<j \end{cases}$$

- $c(i,k)$: Cost of computing A(i..k)
- $c(k+1,j)$: Cost of computing A(k+1..j)
- $p_{i-1}p_kp_j$: Cost of multiplying A(i..j) with A(k+1..j)

All values of k are tried and the best one is chosen

# Application: optimal multiplication of matrices

3. Developing the recurrence relation

$$c(i,j)=\begin{cases} 0 & \text{if } i=j \\ \min\{c(i,k)+c(k+1,j)+p_{i-1}p_kp_j \mid i<=k<j\}, & \text{if } i<j \end{cases}$$

Example

$p0=2$
$p1=20$
$p2=5$
$p3=10$

Only the upper triangular part of the table will be used

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 200 | 300 |
| 2 | - | 0 | 1000 |
| 1 | - | - | 0 |

The elements are computed starting with the diagonal (j-i=0), followed by the computation of elements which satisfy j-i=1 and so on …

# Application: optimal multiplication of matrices

3. Developing the recurrence relation

$$c(i,j)=\begin{cases} 0 & \text{if } i=j \\ \min\{c(i,k)+c(k+1,j)+p_{i-1}p_kp_j \mid i<=k<j\}, & \text{if } i<j \end{cases}$$

Let q=j-i. The table will be filled in for q varying from 1 to n-1

During the computation of c the index of grouping is also stored in a table s.

s(i,j) = k of the optimal grouping of A(i..j)

Algorithm

```
Compute(p[0..n])
 FOR i:=1,n DO c[i,i]:=0 ENDFOR
 FOR q:=1,n-1 DO
   FOR i:=1,n-q DO
     j:=i+q
     c[i,j]:=c[i,i]+c[i+1,j]+p[i-1]*p[i]*p[j]
     s[i,j]:=i
     FOR k:=i+1,j-1 DO
       r:=c[i,k]+c[k+1,j]+p[i-1]*p[k]*p[j]
       IF c[i,j]>r THEN  c[i,j]:=r
                         s[i,j]:=k
       ENDIF
     ENDFOR
   ENDFOR   ENDFOR
 RETURN c[1..n,1..n],s[1..n,1..n]
```

# Application: optimal multiplication of matrices

## Complexity analysis:

Problem size: n

Dominant operation:
multiplication

Efficiency class: $\theta(n^3)$

## Algorithm

```
Compute(p[0..n])
 FOR i:=1,n DO c[i,i]:=0 ENDFOR
 FOR q:=1,n-1 DO
   FOR i:=1,n-q DO
     j:=i+q
     c[i,j]:=c[i,i]+c[i+1,j]+p[i-1]*p[i]*p[j]
     s[i,j]:=i
     FOR k:=i+1,j-1 DO
       r:=c[i,k]+c[k+1,j]+p[i-1]*p[k]*p[j]
       IF c[i,j]>r THEN  c[i,j]:=r
                         s[i,j]:=k
       ENDIF
     ENDFOR
   ENDFOR   ENDFOR
 RETURN c[1..n,1..n],s[1..n,1..n]
```

# Application: optimal multiplication of matrices

1. Constructing the solution

Variants of the problem:

- Find out the minimal number of scalar multiplications
  Solution: this is given by c(1,n)

- Compute A(1..n) in a optimal manner
  Solution: recursive algorithm (opt_mul)

- Identify the optimal groupings (placement of parentheses)
  Solution: recursive algorithm (opt_group)

# Application: optimal multiplication of matrices

Computation of A(1..n) in a optimal manner

Hypothesis: Let us suppose that

- A[1..n] is a global array of matrices (A[i] is $A_i$)
- s[1..n,1..n] is a global variable and classic_mul is a function for computing the product of two matrices.

```
opt_mul(i,j)
  IF i=j THEN RETURN A[i]
  ELSE
      X:= opt_mul(i,s[i,j])
      Y:= opt_mul(s[i,j]+1,j)
      Z:= classic_mul(X,Y)
      RETURN Z
ENDIF
```

# Application: optimal multiplication of matrices

Printing the optimal grouping (the positions where the product is split)

```
opt_group(i,j)
 IF i<>j THEN
          opt_group(i,s[i,j])
          WRITE s[i,j]
          opt_group(s[i,j]+1,j)
 ENDIF
```

# Application: transitive closure of a binary relation

Let $R \subseteq \{1,2,\ldots,n\}x\{1,2,\ldots,n\}$ be a binary relation. Its transitive closure is the smallest (in the sense of set inclusion) relation R* which is transitive and includes R

R* has the following property:

" if i and j are from {1,…,n} and there exists $i_1,i_2,\ldots.i_m$ such that

- $i_1Ri_2, \ldots., i_{m-1}Ri_m$
- $i_1=i$ and $i_m=j$

then i R j"

Examples: R={(1,2),(2,3)}  R*={(1,2),(2,3),(1,3)}

R={(1,2),(2,3),(3,1)}
R*={(1,2),(2,3),(3,1),(1,3),(1,1), (2,1),(2,2),(3,2),(3,3)}

# Application: transitive closure of a binary relation

Even if this is not an optimization problem it can be solved by using the idea of dynamic programming of deriving a recurrence relation.

$R^*$ is successively constructed starting from $R^0=R$ and using $R^1$, $R^2$,… $R^n=R^*$

The intermediate relations $R^k$ (k=1..n) are defined as follows:

$$i\, R^k\, j \iff i\, R^{k-1}\, j \text{ or } i\, R^{k-1}\, k \text{ and } k\, R^{k-1}\, j$$

Example:

$R=\{(1,2),(2,3)\}$ $\qquad$ $R^1=R$

$R^2=\{(1,2),(2,3),(1,3)\}$ $\qquad$ $R^*=R^3=\{(1,2),(2,3),(1,3)\}$

# Application: transitive closure of a binary relation

Representation of binary relations:

Let us consider that a binary relation is represented using a n*n matrix whose elements are defined as follows

$$r(i,j) = \begin{cases} 1 & \text{if } iRj \\ 0 & \text{if not } iRj \end{cases}$$

Example: R ={(1,2),(2,3)}

$$r = \begin{matrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{matrix}$$

# Application: transitive closure of a binary relation

Recurrence relation for the matrices:

$$r^k(i,j) = \begin{cases} 1 & \text{if } r^{k-1}(i,j)=1 \text{ OR } (r^{k-1}(i,k)=1 \text{ AND } r^{k-1}(k,j)=1) \\ 0 & \text{otherwise} \end{cases}$$

Example:

$$r = \begin{matrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{matrix} \quad r1 = \begin{matrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{matrix} \quad r2 = \begin{matrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{matrix} \quad r3 = \begin{matrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{matrix}$$

# Application: transitive closure of a binary relation

Warshall's algorithm

It develops the recurrence relationship on matrices by using two matrices r1 and r2

```
Closure(r[1..n,1..n])
r2[1..n,1..n]:=r[1..n,1..n]
FOR k:=1,n DO
  r1[1..n,1..n]:=r2[1..n,1..n]
  FOR i:=1,n DO
    FOR j:=1,n DO
      IF r1[i,j]=0 OR r1[i,k]=1 AND r1[k,j]=1
        THEN r2[i,j]=1
         ELSE r2[i,j]=0
      ENDIF
    ENDFOR
  ENDFOR
ENDFOR
RETURN r2[1..n,1..n]
```