Dynamic programming
or
the art of avoiding unnecessary computation

Roger Mohr

Nov. 2005

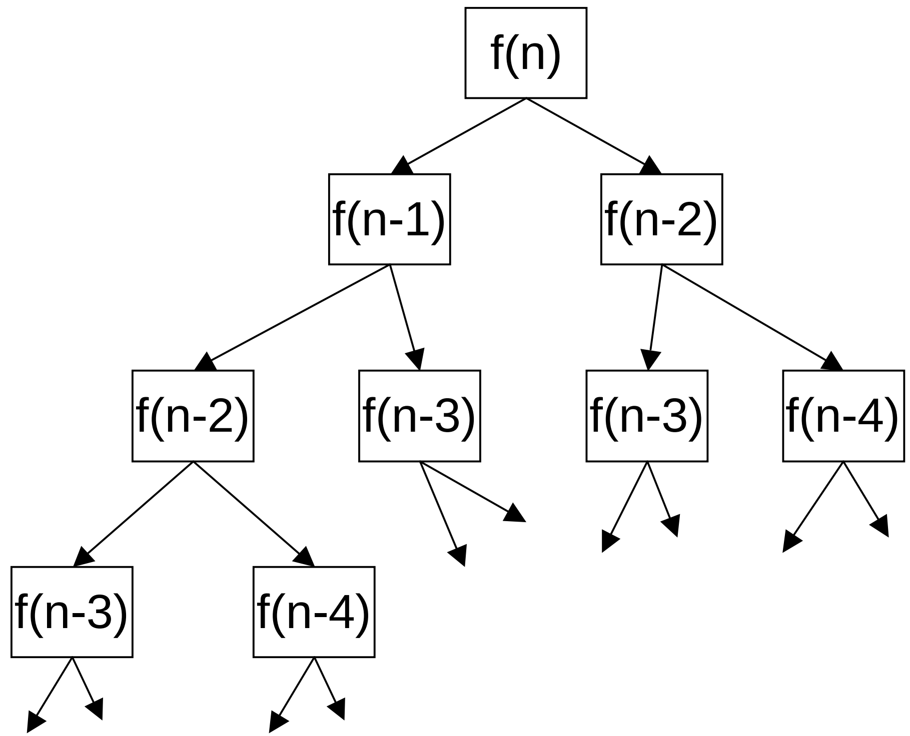
Content

- Two easy initial examples
- General case study: processing an acyclic graph
- A real application
- Conclusion: What to remember

Easy example one : Fibonacci

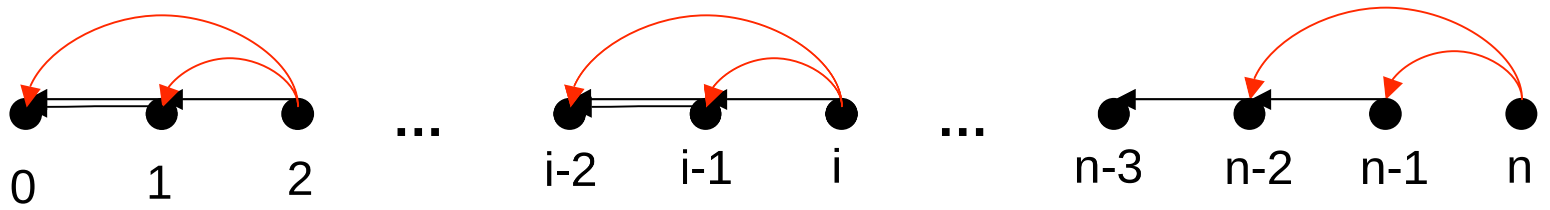
$$\begin{cases} fib(n) = fib(n-1) + fib(n-2) \\ fib(0) = fib(1) = 1 \end{cases}$$

- Direct implementation of this definition leads to :



Redundant computations !
#(f(n-k)) = #(f(n-k+1)) + #(f(n-k+2))

Analysis of what is computed



Conclusion : compute the different value in the ascending order : ➔ $O(n)$

Implementation :

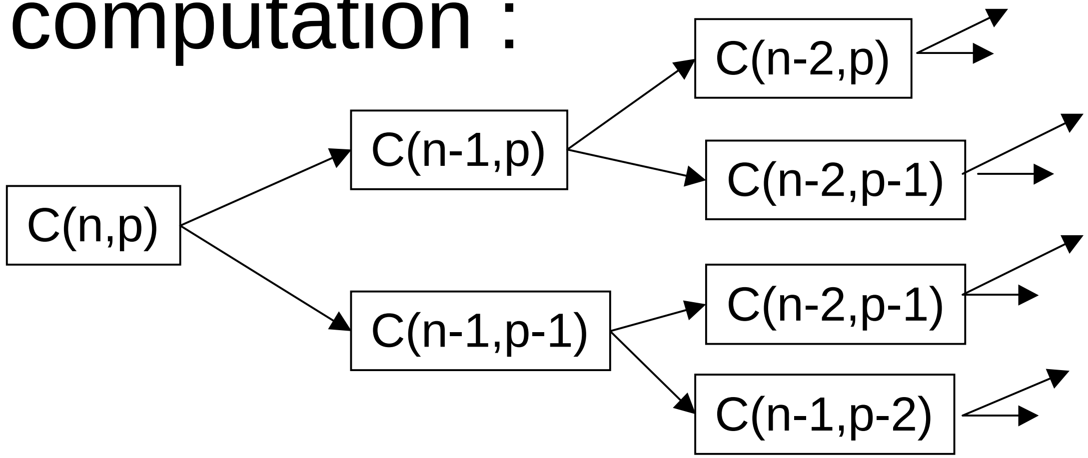
direct : store all the value in an array ➔ space $O(n)$

smart : only 2 values are required :

```
initialisation : f_1:=1; f_2:= 0.
for I in 2..n loop
        xx:= f_1; f_1:= xx+f_2;
        f_2 := xx;
end loop;  -- result is f_1
```

Easy example 2: $\binom{n}{p}$

- Recursive definition:
  - ➢ $C(n,p) = C(n-1,p) + C(n-1,p-1)$
  - ➢ $C(n,n) = 1$; $C(n,0) = 1$
- Direct implementation leads to redundant computation :

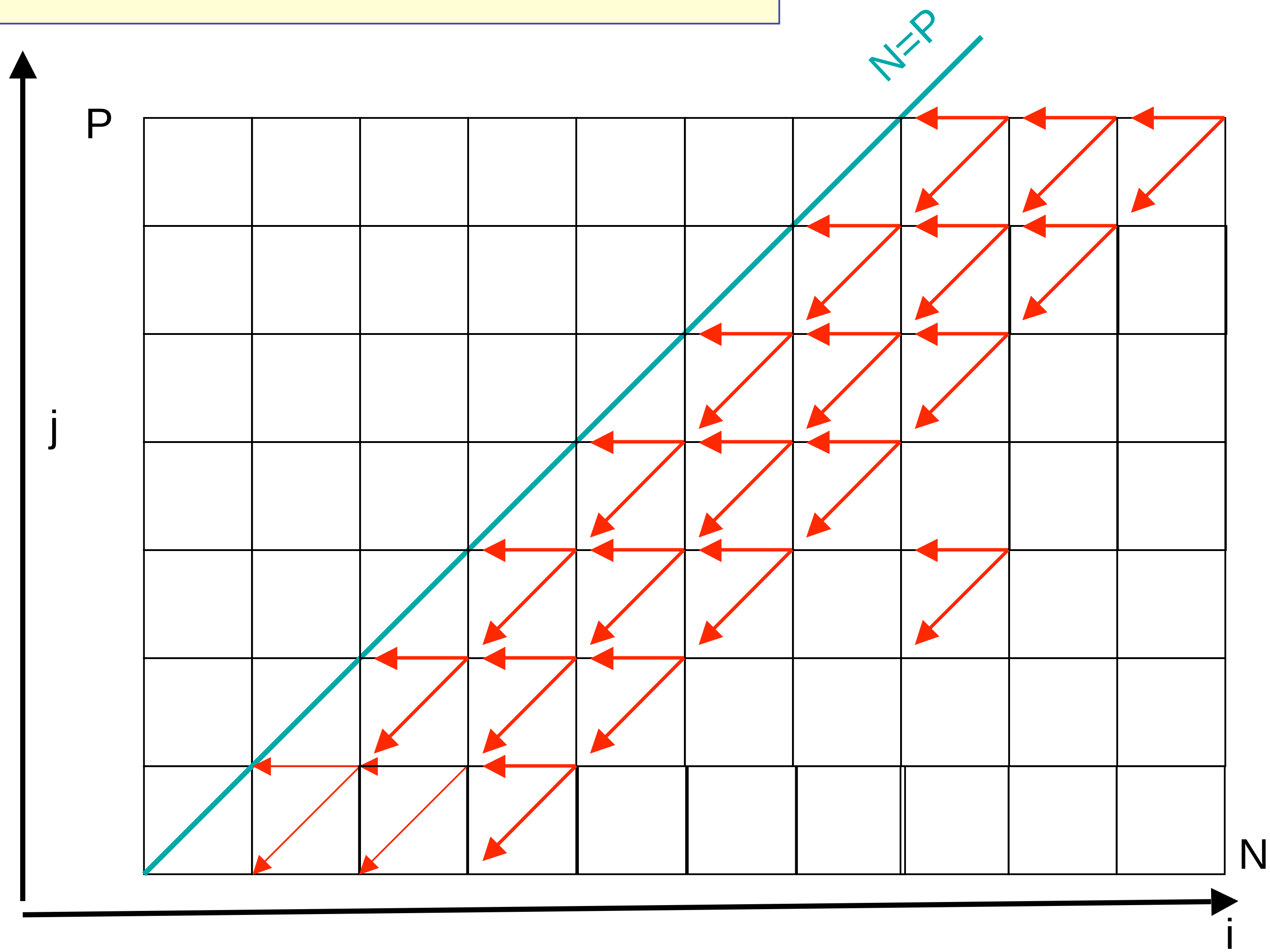


➔ $O(2^n)$

Let us be smart

1. Order

Defined only if p<n



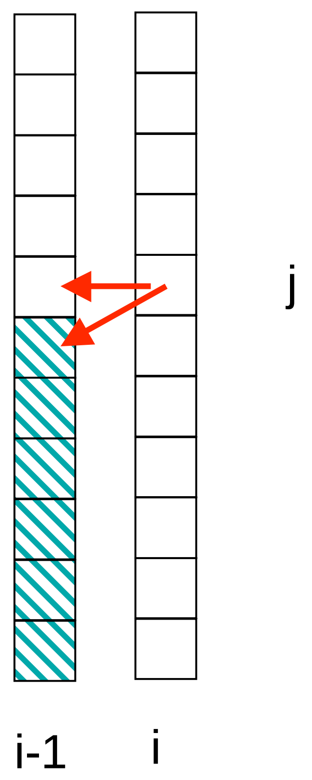
2. implementation

- Direct : store intermediate results in an array C
  - Compute in ascending order
    - Intialisation : C(1,1)=1 ; C(k,0) = 1, k=1..N
    - For I in 2..N loop
    -     If I ≤ P then C(I,I) := 1; end if; -- diagonal term
    -     For J in 1 .. inf (P, I-1) loop
    -         C(I,J) := C(I-1,J) + C(I-1, J-1); end loop
  - Complexity : $O(n^2)$, space : $O(n^2)$,

3. Smart implementation

- First notice that only a subset is required
- Second : As implemented : when computing the $C(I,.)$, only the $C(I-1,.)$ are requested and the previous value $C(I,J-1)$

- Therefore: Only a single vector is required!

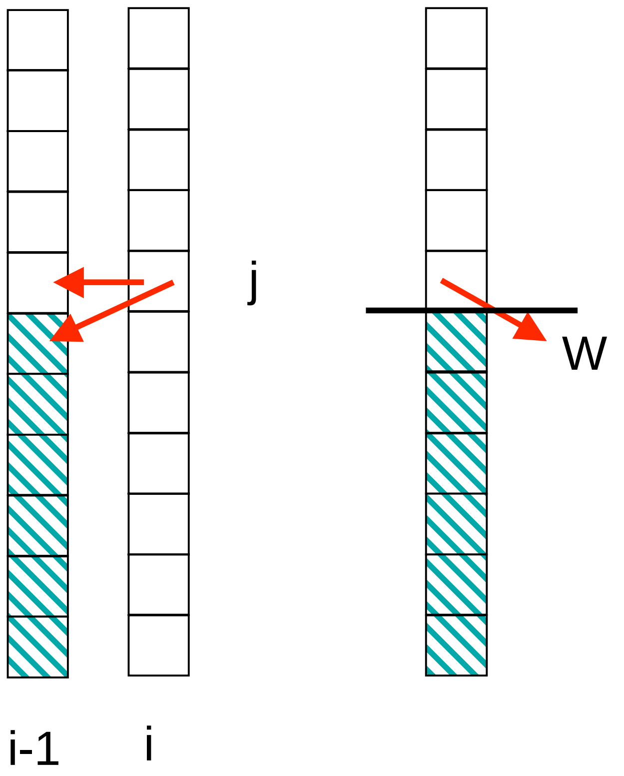


Initialization (left as an exercise)

```
for I in 2 .. n loop          -- for each column i
   if I ≤ P then T(I):=1;       -- diagonal term
         end if;
   W := 1;       -- T(0) = 1
  for J in 1 .. inf(I-1,P) loop
         future_W := T(J);
         T(J) := T(J)+ W;
         W := future_W;
done; done;
```

Space : $O(P)$

W: working variable for temporary storage of T(J-1)



Even smarter

- The whole column don’t need to be computed; computation can be limited for J ranging in max(1, I-(N-P)) .. min(P, I)

- Will lead to time complexity : $O(N\text{x}(N-P))$
- Space can be reduced to a fragment of column : $O(N-P)$

Final code

- Array cell (I,J) will be located at I-J

```
Initialization (left as an exercise)
for I in 2 .. n loop          -- for each column i
   if I ≤ P thenT(0):=1;      -- diagonal term
         end if;
   W := 1;        -- T(0) = 1
  for J in max(1,  I-(N-P)) ..  inf(I-1,P) loop
         Ww := W;
         W := T(J)+ W;
         T(J-1) := Ww;
done; done;
```

Complexity : $O(N \times (N\text{-}P+1))$

The general case: recursive computing in a graph

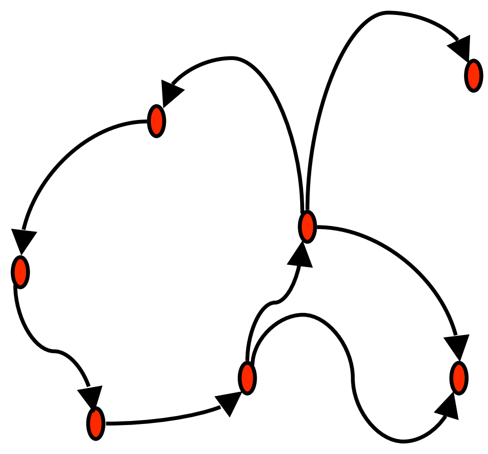
General formula :

$F(x) = \textit{some\_function}\ (F(y) \mid y \in \text{pred}(x))$

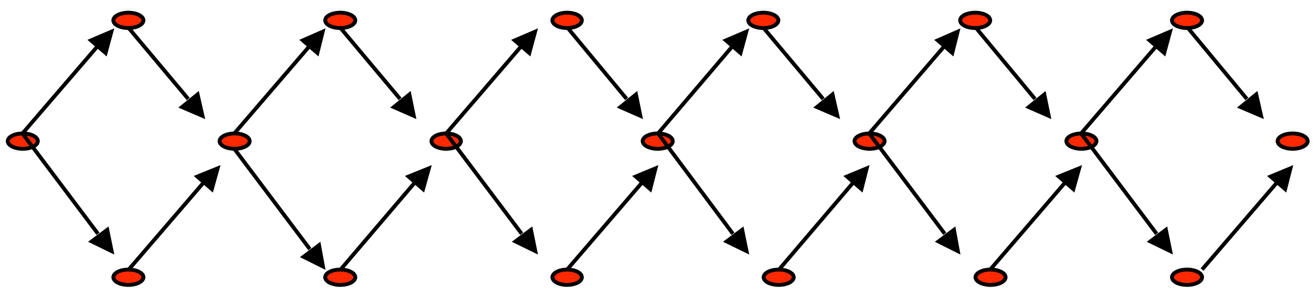
Example : shortest path in a graph

$$\text{dist}(x) = \begin{cases} \text{if } x = \text{start\_node then } 0 \\ \text{else } \min\limits_{y \in \text{pred}(x))} (\text{dist}(y) + \text{cost}\ (x,y)) \end{cases}$$

- General definition valid only if the induction makes sense: no loops in the graph



- Direct implementation can lead to an exponential complexity due to redundant computation



First easy solution

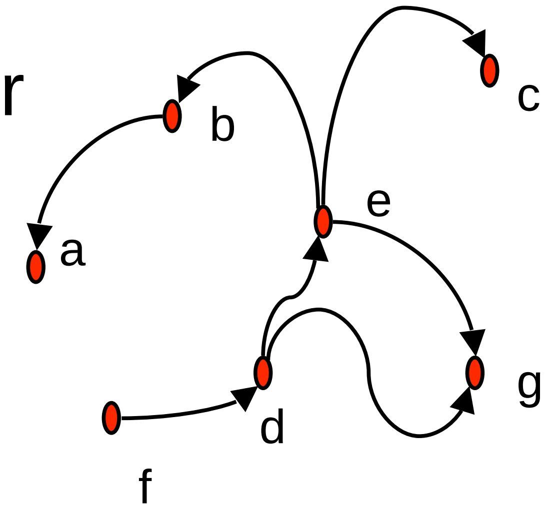
- Store in the array A:
  - A(x) = F(x) if F(x) has been computed,
  - A(x) = @ *otherwise*.

- F(x) =
  - If A(x) ≠ @ then return A(x)
  - else A(x) = *some_function* (F(y)| y ∈ pred(x)) ;
  - return A(x); end if;

Complexity ?

2. Smart solution: Find the right order

- No single solution for a total order compatible with the graph :
  - f, d, e, b, a, c, g   or
  - f, d, e, g, b, a, c   or **….**
- Solution computed through the "topological sorting"
  - Such a sorting solves the problem: computation has to be performed sequentially by following the total order



Topological sorting
(lecture notes: section 7.2.2)

Rule :

- a vertex can be pushed out at the end of the result Res when all his predecessors are already put out

Invariant of the processing loop:

- Res: partial list of the result: the first vertices in the right order

Process :

- Identify a vertex x not in Res with all his predecessors in Res,
- Output x at the end of Res

implementation

- Res : array (1..n) of vertex; L_res : integer:=0
  - *L_res : indexes the last vertex in Res*
- Nbre_Pred: array(1..n) of integer ;
  - *Number of predecessor for each vertex*
- Succ : array (1..n) of vertex_list ;
  - *List of successors for each vertex*

} This encodes the graph

- T_be_P: vertex_list ; -- To_be_Processed
  - *List of vertex waiting to be pushed out because all their predecessors are put out*

algorithm

- Initialization
- While T_be_P ≠ ∅ loop
  - ➢ Extract_one (T_be_P, x);

    Here you might perform *some_function* in order to compute F(x)
  - ➢ L_Res := L_Res + 1; Res(L_Res):= x;
  - ➢ For all y in Succ(x) loop
    - Nbre_Pred(y) := Nbre_Pred(y) – 1;
    - If Nbre_Pred(y) = 0 then add_to_list (y, T_be_P)

      end if;
    - done
  - ➢ done;

Algorithm (cont.)

- Initialization :
  - T_be_P := empty_list;
  - For all vertex z loop
    - If Nbre_Pred(z) = 0 then add_to_list (y, T_be_P) end if;

- Complexity :
  - Each edge is visited as most once : $O(m)$

  (m = nbr of edges)

3- A real application: string distance

- Problem :
  - 2 input strings: $\alpha = a_1 \ldots a_n$, and $\beta = b_1 \ldots b_m$,
  - Output : the distance between α and β

- Application: measuring similarity between words in speech recognition (other applications in image processing, etc.)

- Distance : minimum cost of transforming α into β
  - del (a) : cost of deleting character a
  - ins (a) : cost of inserting character a
  - chg (a,b) : cost of changing a into b (might be 0 if a=b)
  - Total cost between α and β : minimal sum of costs allowing to transform α into β

Step 1: stating the problem

- $d(aX, bY) = \min(chg(a,b)+d(X,Y),$
  $del(a) + d(X,bY),$
  $ins(b) + d(aX,Y))$
- $d(\varepsilon, bY) = ins(b) + d(\varepsilon, Y)$
- $d(aX, \varepsilon) = del(a) + d(X, \varepsilon)$
- $d(\varepsilon, \varepsilon) = 0$

$d(a_i a_{i+1} .. a_n,\ b_j b_{j+1} \ldots b_m)$ might be referred by the subscripts (i, j)

- Leads to an obvious recursive program, with redundant computations (left as an exercise) and exponential complexity (left as an exercise; see lecture notes too)

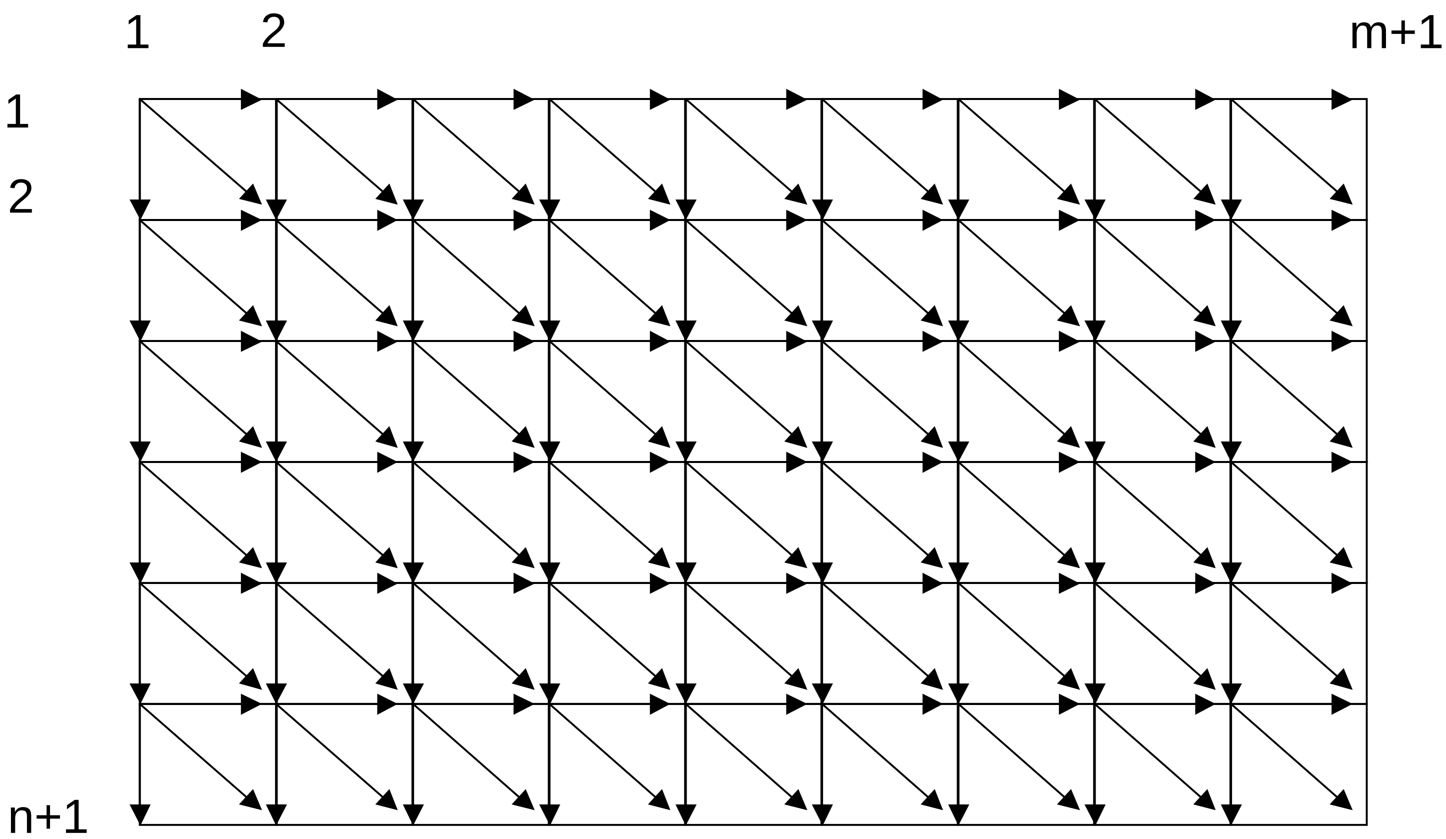
The easy solution

- Array C(i,j) :
  - C(i,j) = d(a_i..a_n, b_j…b_m) if this value is computed
  - C(i,j) = @ otherwise

- Direct implementation from the definition

```
d(i,j) = if C(i,j) ≠ @ then return C(i,j)
         else C(i,j) = min (chg(a(i),b(j)) + d(i+1,j+1),
                            del(a(i)) + d(i+1, j),
                            ins(b(j)) + d(j, +1) );
             return C(i,j); end if;
```

Smart solution: analysis of the order



- Direct iterative implementation with an array C(i,j) storing $d(a_i..a_n, b_j…b_m)$:

- Intialization: d(n+1,m+1) = 0
  - For J in reverse 1..m loop

    C(n+1,j) = ins(b(J)) + C(n+1,J+1); loop;
  - For I in reverse 1..n loop

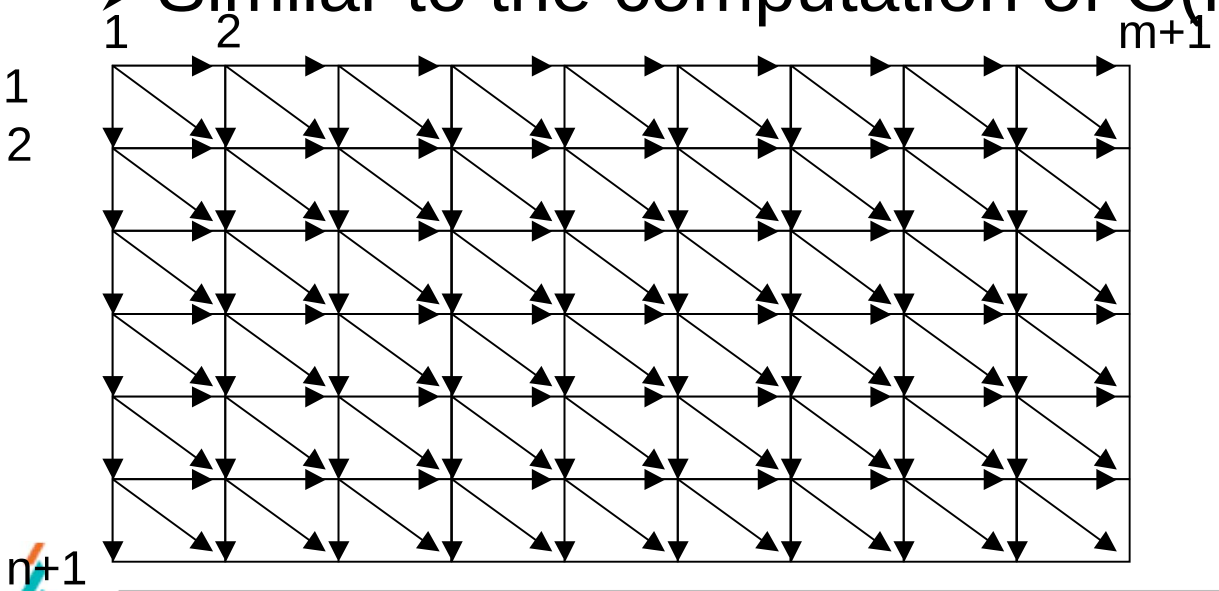
    C(I,m+1) = del (a(I)) + C(I+1, m+1); loop;
- General loop :
  - For I in reverse 1..n loop; for J in reverse 1..m loop

    C(I,J) = min((chg(a(i),b(j)) + C(i+1,j) -- from the initial definition

    …);
  - end loop; end loop;
- Final result : C(1,1)

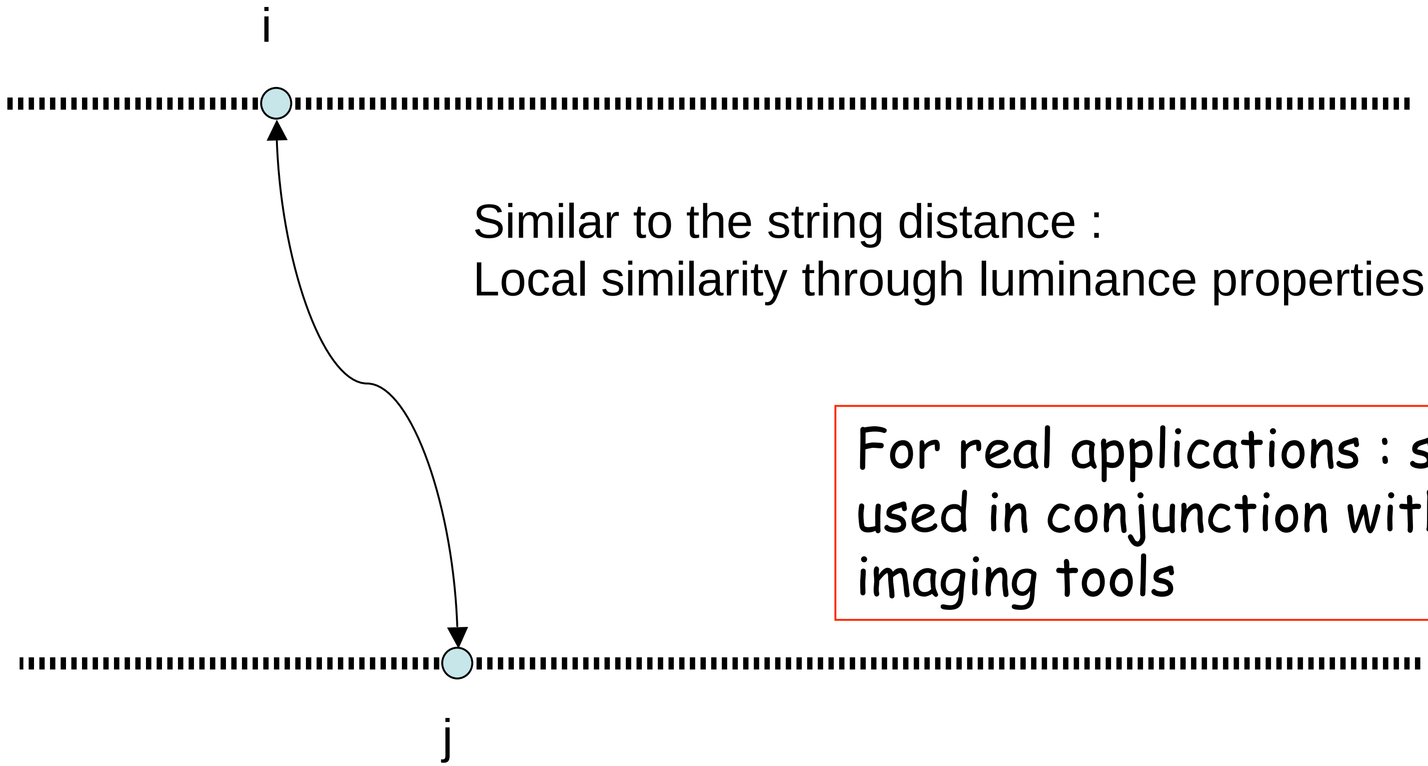
- Complexity : $O(n \times m)$
- Space complexity : $O(n \times m)$

- Reduction of space complexity to $O(n)$
  - Similar to the computation of C(n,p)



Application to a pair of stereo images

Matching along the epipolar lines



Similar to the string distance :
Local similarity through luminance properties

For real applications : sometimes used in conjunction with other imaging tools

Lessons to remember: five steps

1. Get the right recursive definition first
2. Check if redundant computations happen (easy)
3. If so: see if storing intermediate computation in a global array will solve the complexity problem (*usually easy*)
4. Derive an iterative program : Analyze the calculus order and check how to compute the intermediate results in order
5. (*optional*) is the space of the array really required or can it be reduced?

Counter examples

- no redundant computation: N-queen, coloring graph, … exponential and cannot be reduced!

- Ackerman function:

$$A(n,m) = \begin{cases} n=0 \rightarrow m+1 \\ m=0 \rightarrow A(n-1,1) \\ \text{others} \rightarrow A(n-1,A(n,m-1)) \end{cases}$$

Obviously redundant computation: why can they not be stored in an array? (hint: apply the fact that n→A(n,n) is a functions which grows faster than any known standard function)