Dynamic Programming

Algorithms Design Paradigms

- Divide and Conquer
  - Quick Sort;   Binary Search; ....
- Greedy
  - Where a divide phase doesn't work !
  - Choose the best possible recourse leading to the optimality
- Dynamic Algorithms
  - Where a divide phase doesn't work!
  - Use backtracking when a selection doesn't turn to be optimal.

Algorithms Design Paradigms (contd)

- Greed
  - Build up a solution incrementally, myopically optimizing some local criterion.
- Divide-and-conquer
  - Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming
  - Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
  - a powerful technique that can be used
    - to solve many problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time.

Introduction to Dynamic Programming

- "Those who do not remember the past are condemned to repeat it"
  - has evolved into a major paradigm of algorithm design in CS.
- How did the term originate?
  - In 1957, Richard Bellman coined it as a method of solution
  - to describe solution to a type of an optimum control problem.

Etymology

- Dynamic
  - is something that depends on the current state.
- Programming imply a series of choices
  - statically programmed radio programs vis-à-vis phone-in programs.
- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
  - Bellman sought an impressive name to avoid confrontation.
  - "it's impossible to use dynamic in a pejorative sense"
  - "something not even a Congressman could object to"
- Top-down design vrs Bottom-up design
  - Dynamic Progg may take either of the Bottom-up OR the Top-Down approach – though more logical is Bottom-up

Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ….

- Some famous dynamic programming algorithms.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.
  - Smith-Waterman for sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.

Introduction to Dynamic Programming(contd)

- Divide & Conquer – we divide a problem into smaller instances and attempt to solve those .
- Dynamic Progg also does the same
  - i.e. typically provides many ways to divide the original problem into subproblems.
  - However, it is not evident which division leads to solution
  - then, how can this approach work ?
  - this approach works because dynamic programming solves all subproblems whose solutions that might be potentially needed.
  - the landmark of DP is that it most of the times converts a exponential algorithm to a polynomial algorithm.

Introduction to Dynamic Programming(contd)

- On the other hand it is
  - opposite of the greedy approach
    - implicitly explores all the possible solutions
  - is useful in those applications
    - which require solution to all the subproblems that may be used in framing the eventual optimal solution
    - e.g.…………..
  - careful decomposition of a problem into subproblems
    - avoids any repetition
    - builds solutions to larger & larger subproblems
  - is dangerously close to the brute-force approach
    - but, then is it advisable ?

Three aspects of focus, now

- To understand how Top-down-design – like in recursion is natural and powerful – but if not controlled properly, it can become very inefficient.
- How dynamic progg also solves the problems of smaller sizes – the subproblems, first; but sort of follows a bottom up approach – stores the solutions and later looks up them up.
- How to characterize the dynamic programming algorithms – to get a unified framework using which a recrusive algorithm can be converted into a dynamic programing one.

Introduction to Dynamic Programming (contd)

- "Top-down design is natural, powerful
  - but what about the efficiency concerns?"
- Recursion if not controlled properly…………
- Often compilers do not do justices to such top-down designed algorithms
  - need to provide more information to the compilers
- How?
  - Using a table of choices instead of recursion.
  - e.g. a language like Haskell

Introduction to Dynamic Programming (contd)

- Underlying idea
  - Avoid duplication of efforts
  - Use a table where solutions of subinstances are stored
  - implicitly explore the space of all possible solutions
    - but without ever examining them all explicitly

An illustration – Fibonacci series

- Calculate Fibonacci numbers

```
int fib-recurs(int n) {
  if (n < 2) return n;
  else
    return fib(n-1) + fib(n-2);
}
```

  - Simple, elegant!
  - But .... Let’s analyze it’s performance!

Analysis - Fibonacci series

- Analysis
  - If time to calculate Fib(n) is t_n
    then $t_n = t_{n-1} + t_{n-2}$
  - Now $t_1 = t_2 = c$ *i.e.* $T(1)=T(2)=c$
  - So $t_n = c'$ Fib(n-2) = $O(2^n)$

```
int fib( int n ) {
  if ( n < 2 ) return n;
  else
    return fib(n-1) + fib(n-2);
}
```

Analysis (contd)

- Prove that time to calculate F(n) is $O(2^n)$.
- Recollect that F(0) = 0 and F(1) = 1 and F(n) = F(n-1) + F(n-2)
- What is the recurrence relation for recursive Fibonacci ?
  - T(n) = T(n-1) + T(n-2) + time required for addition i.e. O(1)
- Thus, the statement is now prove that the solution to the recurrence T(n) = T(n-1) + T(n-2) + time required for addition i.e. O(1) is $O(2^n)$.
- Solving by mathematical induction.
  - Basis: n=1, does recurrence solve to $O(2^1)$?
  - IH: Assume that $T(n-1) = O(2^{n-1})$,
  - To prove: $T(n)=O(2^n)$

- Actually, $\mathbf{f_{n+1} = O(1.618^n)}$

  this is definitely not an efficient algorithm!!

Simulation

- To appreciate the root-cause of inefficiency
  - draw the call-graph for fib-recurs(6)
- What is problematic in the call-graph ?
- What is the minimum running time required ?
- So the issue is
  - Is it not possible to compute F_n with $\theta(n)$ simple statements (which involve no further calls) and remembering n smaller values?
- How about the iterative solution to the problem?

Fibonacci series – Iterative approach

```
int fib( int n ) {
  int f1, f2, f;
    if ( n < 2 ) return n;
  else {
    f1 = f2 = 1;
    for( k = 2; k < n; k++ ) {
      f = f1 + f2;
      f2 = f1;
      f1 = f;
      }
  return f;
}
```

Fibonacci series – Iterative approach

```
int fib( int n ) {
   int f1, f2, f;
     if ( n < 2 ) return n;
  else {
    f1 = f2 = 1;
    for( k = 2; k < n; k++ ) {
      f = f1 + f2;
      f2 = f1;
      f1 = f;
      }
  return f;
}
```

Note the `f1`, `f2` here

We start by solving the smallest problems

Then use those solutions to solve bigger and bigger problems

Fibonacci series – Iterative approach

- Exercise:
  - draw the call structure for the above.
  - Is it a tree of a DAG ?
- Which approach 've we followed here?
  - Bottom up or top down?

Dynamic Programming – Basic Paradigm

- Dynamic Approach
  - Solve the small problems
  - Store the solutions
  - Use those solutions to solve larger problems
- But, what does dynamic programming approach require additionally ?
  - space
- Compare the time complexities of the Iterative Fibonacci with Recursive Fibonacci.
  - $O(n)$ *vrs* $O(n^n)$ for the recursive case!

Basic Paradigm (contd)

- It is clear that
  - the growth of the redundant calculations is explosive in recursive algorithms.
- Why can't a compiler handle a recursive program the same way ? i.e.
  - by keeping a stored list of precomputed values and NOT making a recursive call each time.
- But, then this efficiency is a "borrowed" one and
  - NOT inherently built into the algorithmic design approach.

Binomial Coefficients

Problem: Binomial Coefficients

- A binomial coefficient, denoted *C(n, k),* is the number of combinations of *k* elements from an *n*-element set *(0 ≤ k ≤ n).*
- That is……….

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \text{ for } 0 < k < n$$

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < \mathrm{k} < \mathrm{n} \\ 1 & k = 0 \text{ or } k = n \end{cases}$$

$$\binom{n}{0} = 1 \qquad \binom{n}{n} = 1$$

Problem: Binomial Coefficients

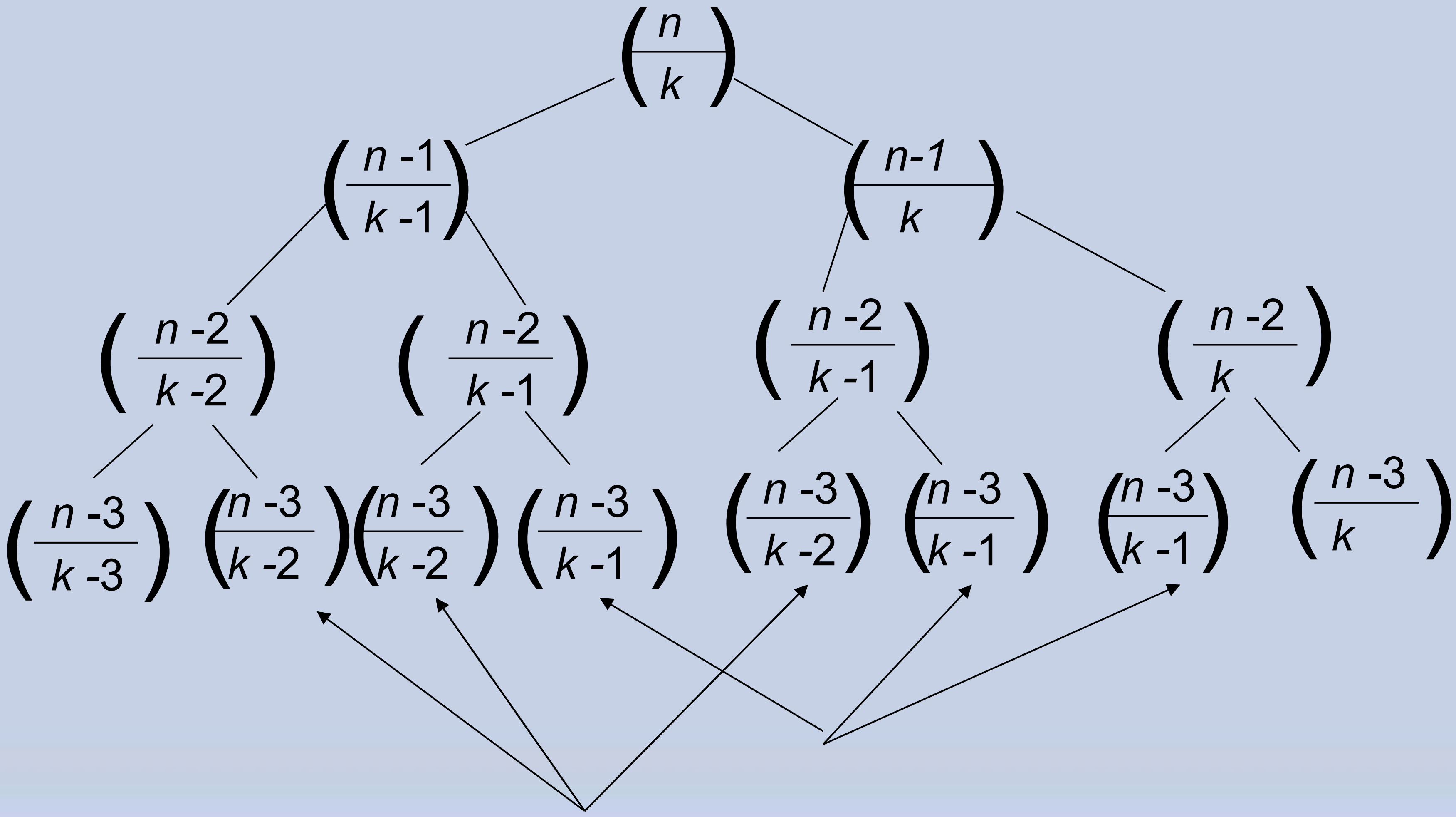
- We can calculate it directly by the function below

```
Algorithm C(n,k)
if k = 0 or k = n
   then return 1
else return C(n-1, k-1)+ C(n-1, k)
```

- The recurrence relation (a problem → 2 overlapping subproblems) would be given by……
  - $C(n, k) = C(n-1, k-1) + C(n-1, k)$, for $n > k > 0$, and
  - $C(n, 0) = C(n, n) = 1$

Analysis

- If is apparent that many values of **C(i j) i< n, j<k** are calculated over and over again………



Dynamic Solution

- Use a matrix B of n+1 rows, k+1 columns where

$$\mathbf{B[n, k]} = \binom{n}{k}$$

- *Establish* a recursive property. Rewrite in terms of matrix B:

$$\mathbf{B}[\,i\,,\,j\,] \begin{cases} = \mathbf{B}[\,i - 1\,,\,j - 1\,] + \mathbf{B}[\,i - 1,\,j\,] & ,0 < j < i \\ = 1 & ,\,j = 0 \text{ or } j = i \end{cases}$$

- Solve all "smaller instances of the problem" in a *bottom-up* fashion by computing the rows in *B* in sequence starting with the first row.

Compute B[4,2]= $\binom{4}{2}$

- Row 0: **B[0,0] =1**
- Row 1: **B[1,0] = 1**
  B[1,1] = 1
- Row 2: **B[2,0] = 1**
  B[2,1] = B[1,0] + B[1,1] = 2
  B[2,2] = 1
- Row 3: **B[3,0] = 1**
  B[3,1] = B[2,0] + B[2,1] = 3
  B[3,2] = B[2,1] + B[2,2] = 3
- Row 4: **B[4,0] = 1**
  B[4,1] = B[3, 0] + B[3, 1] = 4
  B[4,2] = B[3, 1] + B[3, 2] = 6

Alternate Approach

- Instead, a table of intermediate values – as

1 k=0

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1 k=5

n=0 n=5

⇑ing n values

⇑ing k values

- Each entry O(1) time and there are O(n^2) entries
- Therefore, O(n^2) time to calculate B.C.

Dynamic Program

```
Binomial (n,k )
for i  = 0 to n // every row
    for j = 0  to minimum( i, k )
        if j = 0 or j = i   // column 0 or diagonal
            then B[ i , j ] = 1
            else  B[ i , j ] =B[i -1, j -1]  +  B[i -1, j ]
return B[ n, k ]
```

Dynamic Program

- What is the run time?
- How much space does it take?
- If we only need the last value, can we save space?
- All values in column 0 are 1
- All values in the first $k+1$ diagonal cells are 1
- $j \neq i$ and $0 < j \leq \min\{i,k\}$ ensures we only compute B[i, j] for $j < i$ and only first $k+1$ columns.

Number of iterations

$$\sum_{i=0}^{n}\sum_{j=0}^{\min(i,k)}1=\sum_{i=0}^{k}\sum_{j=0}^{i}1+\sum_{i=k+1}^{n}\sum_{j=0}^{k}1=$$

$$\sum_{i=0}^{k}(i+1)+\sum_{i=k+1}^{n}(k+1)=$$

$$\frac{(k+2)(k+1)}{2}+(n-k)(k+1)=$$

$$\frac{(2n-k+2)(k+1)}{2}$$

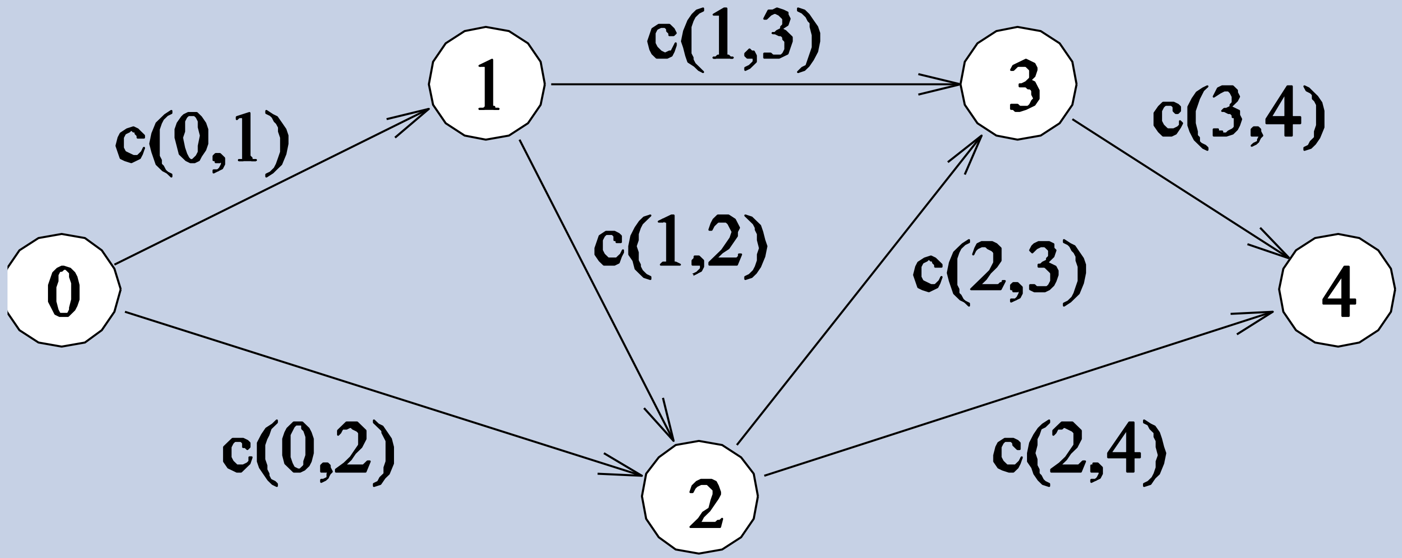
Shortest Paths in DAGs

DP: Shortest Paths in DAGs

- Consider the problem of finding a shortest path between a pair of vertices in an acyclic graph.
- An edge connecting node *i* to node *j* has cost *c(i,j)*
- The graph
  - contains *n* nodes numbered *0,1,…, n-1*, and
  - has an edge from node *i* to node *j* only if *i* < *j.*
  - Node 0 is source and node *n-1* is the destination.

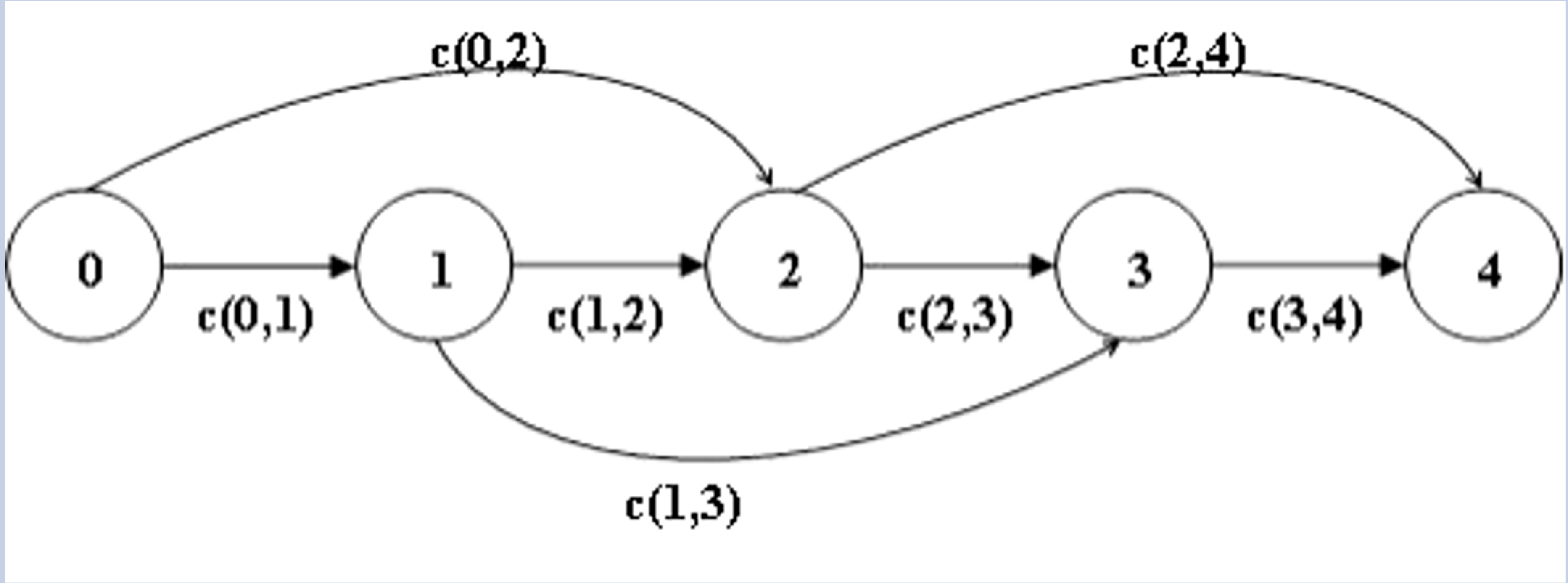
DP: Shortest Paths in DAGs

- The nodes of a DAG can be linearized
  - e.g. for the graph show below, what would be the linearized DAG



DP: Shortest Paths in DAGs

- The nodes of a DAG can be linearized
  - e.g. for the graph show above, the linearized DAG would be as follows



DP: Shortest Paths in DAGs

- The advantages are :
  - It is easy to compute the shortest distances from a source node to any other node in the graph e.g.
  - `dist(4) = min{dist(3) + c(3,4), dist(2) + c(2,4)}`
  - such relation can be written for every node in the DAG
    - by the time we reach a particular node, we already have the "DP table" information we need to compute the shortest distance to that node.
  - hence, we are able to compute all distances in a single pass

DP: Shortest Paths in DAGs

- Let $f(x)$ be the cost of the shortest path from node 0 to node x.

$$f(x) = \begin{cases} 0 & x = 0 \\ \min\limits_{0 \le j < x} \{f(j) + c(j, x)\} & 1 \le x \le n-1 \end{cases}$$

DP: Shortest Paths in DAGs

Algorithhm SPDAG

```
•   initialize all dist(.) values to infinity
•   dist(s)=0
1.  for each v ∈ V\{s}, in linearized order:
2.      dist(v) = min_{u,v∈E}{dist(u) + l(u,v)}
```