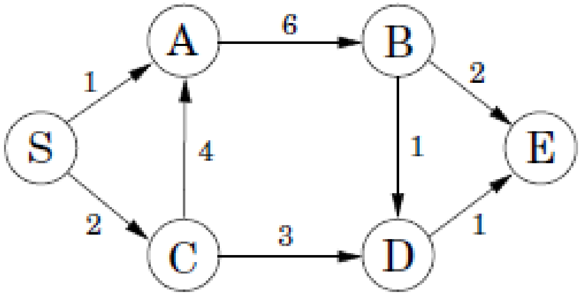
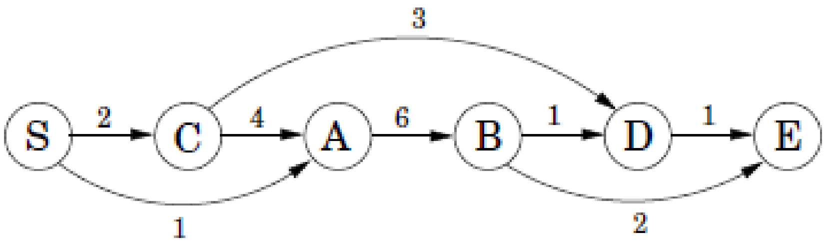
Dynamic programming

Shortest paths in dags

- A dag and its linearization



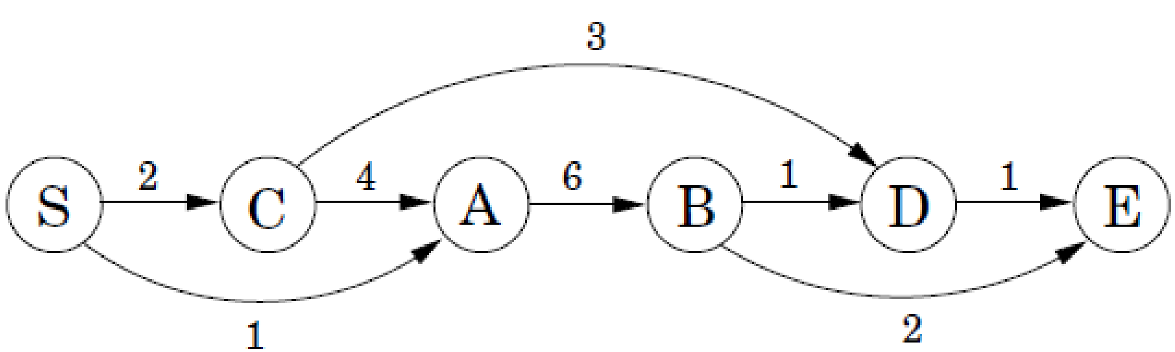


- To compute the distance from S to D, only need to consider distance to C and to B (because B and C are two predecessors to D).
- dist (D) = min { dist(B) + 1 , dist(C) + 3 }
- If we compute dist values in the left-to-right order, we can make sure that when we get to a node *v*, we already have all the information we need to compute dist(*v*).

Shortest paths in dags

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

- The algorithm solves a collection of subproblems, $\{\mathrm{dist}(u) : u \in V\}$.
- Starting with dist(s), then solve "larger" subproblems.

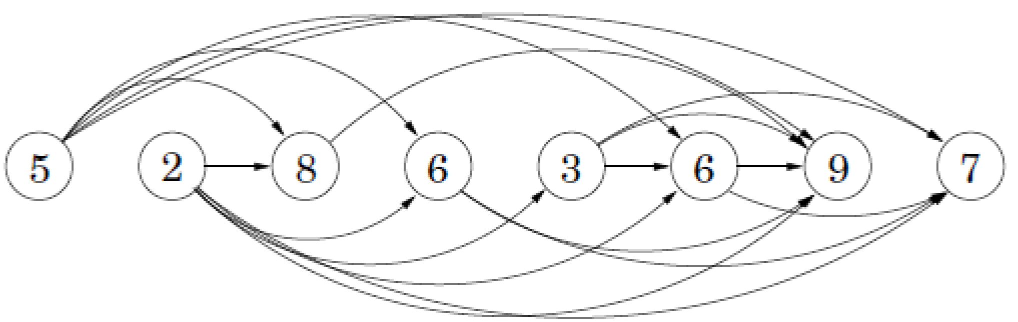


Dynamic programming

- a very powerful algorithmic paradigm
- a problem is solved by identifying a collection of *subproblems* and tackling them one by one
  - smallest first
  - using the answers to small problems to solve larger ones,
  - until the original problem is solved.

Longest increasing subsequences

- Input : a sequence of numbers $a_1, \ldots, a_n$
- A *subsequence* is any subset of these numbers taken in order, of the form $a_{i1}, a_{i2}, \ldots, a_{ik}$ where $1 \le i_1 < i_2 < \ldots < i_k \le n$
- Goal : to find the increasing subsequence of greatest length.
- E.g.) the longest increasing subsequence of 5, 2, 8, 6, 3, 6, 9, 7 :
  - 2, 3, 6, 9



- Find *the longest path* in the dag!

Longest increasing subsequences

- $L(j)$: the length of the longest path – the longest increasing subsequence – ending at j

- Algorithm

```
for j = 1, 2, ..., n:
    L(j) = 1 + max{L(i) : (i, j) ∈ E}
return max_j L(j)
```

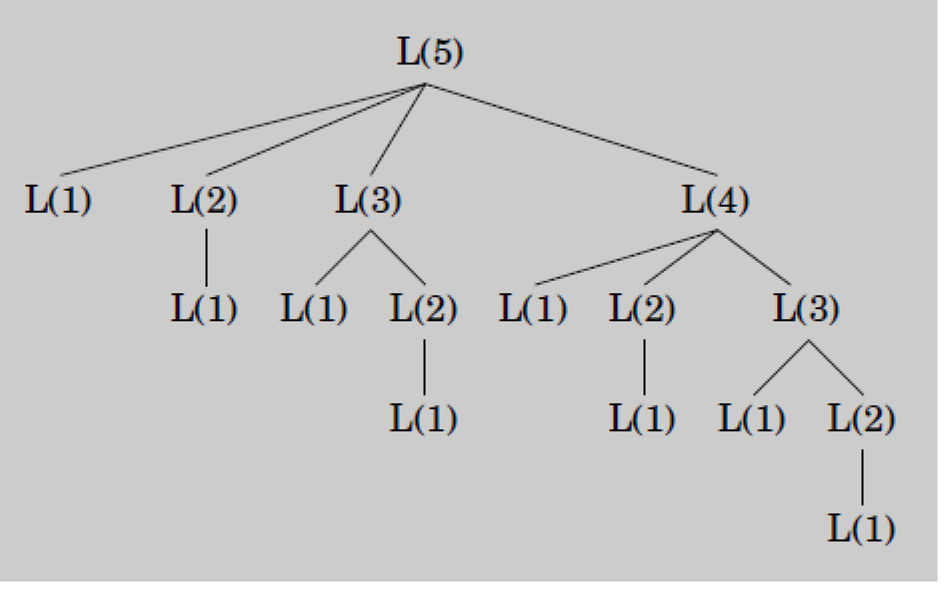
- *Dynamic programming* : To solve the original problem, define a collection of *subproblems* { $L(j)$: $1 \le j \le n$ } with the key property (*):
  – (*) There is an *ordering* on the subproblems,
    and a *relation* that shows how to solve a subproblem
    given the answers to "smaller" subproblems
    (subproblems that appear earlier in the ordering).

Longest increasing subsequences

- Each subproblem is solved using the relation :
  - $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$
- How long does this step take?
  - To compute $L(j)$: O(in-degree(j)).
  - Total : $O(|E|) \rightarrow O(n^2)$.
- L values only tells us the *length* of the optimal subsequence. How to construct the subsequence?
  - While computing $L(j)$, record prev(j), the previous node on the longest path to j.

Recursive vs. dynamic programming

- The formula for $L(j)$ suggests an alternative, *recursive* algorithm.
- Suppose that the numbers are sorted. Then, $L(j) = 1 + \max \{ L(1), L(2), \ldots, L(j-1)\}$.
- The following figure unravels the recursion for $L(5)$:



- The tree for $L(n)$ has *exponential* size. Many *repeated* nodes!
- Only *small* number of *distinct* subproblems -> DP solve them in the right order.

<u>Edit distance</u>

- Given two strings, how can we measure how close they are?
- Ex) SNOWY, SUNNY : possible alignments

S	—	N	O	W	Y
S	U	N	N	—	Y

Cost: 3

—	S	N	O	W	—	Y
S	U	N	—	—	N	Y

Cost: 5

- - : gap (we may place any number of gaps in either string)
- Cost : the number of columns in which the letters differ
- *Edit distance* of two strings : the cost of their best possible alignment

  = minimum number of *edits* – insertions, deletions, and substitutions of characters – needed to transform the first string into the second

Dynamic programming

- *What are the subproblems*?
- (*) There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems (subproblems that appear earlier in the ordering).
- Input : $x[1..m]$, $y[1..n]$
- Consider prefixes : $x[1..i]$, $y[1..j]$ -> call this subproblem $E(i, j)$
- Subproblem $E(7, 5)$

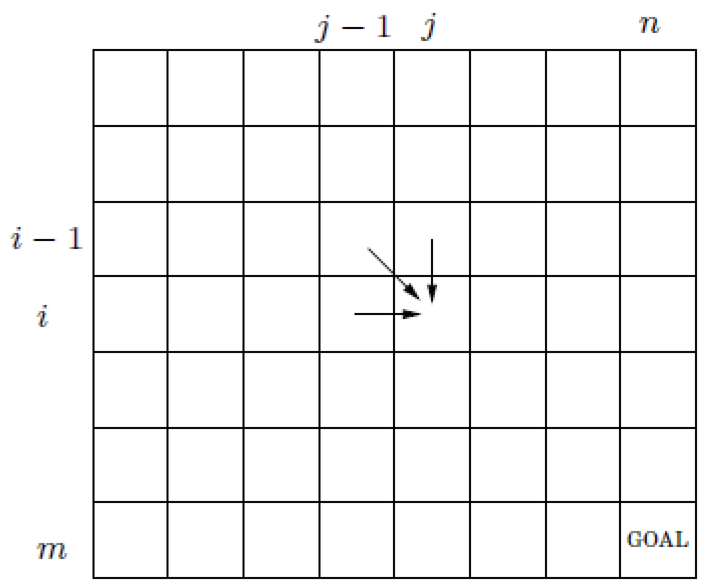
$$\boxed{\text{E X P O N E N}}\ \text{T I A L}$$

$$\boxed{\text{P O L Y N}}\ \text{O M I A L}$$

- Goal : $E(m, n)$

- Express $E(i, j)$ in terms of smaller subproblems!
- The rightmost column of the best alignment can be one of the following : $\begin{array}{c} x[i] \\ - \end{array}$ **or** $\begin{array}{c} - \\ y[j] \end{array}$ **or** $\begin{array}{c} x[i] \\ y[j] \end{array}$
- $E(i, j) = \min \{1 + E(i\text{-}1, j), 1 + E(i, j\text{-}1), \text{diff}(i, j) + E(i\text{-}1, j\text{-}1) \}$

  where $\text{diff}(i, j) = 0$ if $x[i] = y[j]$ and 1 otherwise.
- Base cases : i=0 or j=0
- The answers to all the subproblems $E(i, j)$ form a two-dimensional table.



$\texttt{for}\ \ i = 0, 1, 2, \ldots, m\texttt{:}$

$\quad E(i, 0) = i$

$\texttt{for}\ \ j = 1, 2, \ldots, n\texttt{:}$

$\quad E(0, j) = j$

$\texttt{for}\ \ i = 1, 2, \ldots, m\texttt{:}$

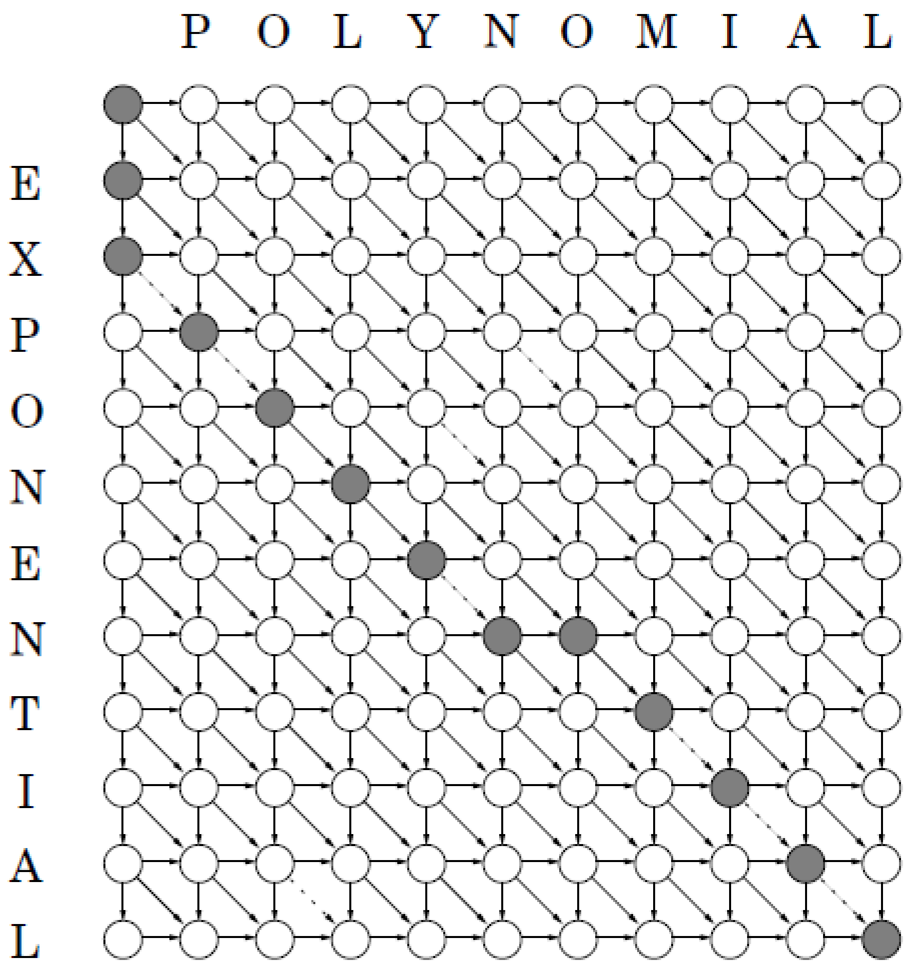
$\quad \texttt{for}\ \ j = 1, 2, \ldots, n\texttt{:}$

$\qquad E(i, j) = \min\{E(i-1, j) + 1, E(i, j-1) + 1, E(i-1, j-1) + \texttt{diff}(i, j)\}$

$\texttt{return}\ \ E(m, n)$

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Underlying dag



Common subproblems

- Finding the right subproblem takes creativity and experimentation.
- Standard choices

i. The input is $x_1, x_2, \ldots, x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is therefore linear.

ii. The input is $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

$$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$$

The number of subproblems is $O(mn)$.

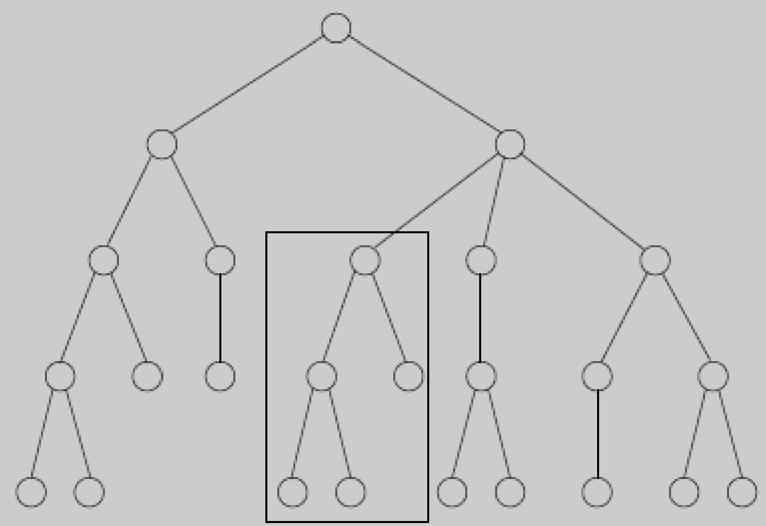
Common subproblems

iii. The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$.

$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is $O(n^2)$.

iv. The input is a rooted tree. A subproblem is a rooted subtree.



Knapsack

- Given a knapsack of capacity W, n items of weight $w_1,\ldots, w_n$ and value $v_1 ,\ldots, v_n$, choose the most valuable combination of items.
- E.g.) W=10

Item	Weight	Value
1	6	$30
2	3	$14
3	4	$16
4	2	$9

- Two versions :
  - 1) allow unlimited quantities :

    pick item 1 and two of item 4 (total $48)
  - 2) allow only 1 of each item :

    pick items 1 and 3 (total $46).

Knapsack with repetitions

- What are the subproblems?
- Define $K(w)$ = maximum value achievable with a knapsack of capacity w.
- If the optimal solution to $K(w)$ includes item i, then removing it leaves an optimal solution to $K(w-w_i)$.
- We don't know which i, so try all possibilities.

$$K(w) = \max_{i: w_i \le w} \{K(w - w_i) + v_i\}$$

- Algorithm

$K(0) = 0$
for $w = 1$ to W:
    $K(w) = \max\{K(w - w_i) + v_i : w_i \le w\}$
return $K(W)$

Analysis

$K(0) = 0$
for $w = 1$ to W:
  $K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\}$
return $K(W)$

- This algorithm fills in a one-dimensional table of length *W*+1, in left-to-right order.
- Each entry can take up to O(*n*) time to compute.
- The overall running time = O(*nW*).

Knapsack without repetition

- Need to refine the subproblem to carry additional information about the items being used by adding a second parameter, $0 \leq j \leq n$.
- $K(w, j)$ = maximum value achievable using a knapsack of capacity w and items $1, \ldots, j$.
- Goal : $K(W, n)$.
- Express $K(w, j)$ in terms of smaller subproblems considering whether item j is needed or not.
- $K(w, j) = \max \{ K(w-w_j, j-1) + v_j , K(w, j-1) \}$
- (The first case is invoked only if $w_j \leq w$)

Analysis

```
Initialize all K(0, j) = 0 and all K(w, 0) = 0
for j = 1 to n:
    for w = 1 to W:
        if w_j > w:   K(w, j) = K(w, j − 1)
        else:   K(w, j) = max{K(w, j − 1), K(w − w_j, j − 1) + v_j}
return K(W, n)
```

- The algorithm fills out a 2-dimensional table, with W+1 rows and n+1 columns.
- Each table entry takes constant time.
- Running time : O(nW).

Memoization

- In dynamic programming, we use a recursive formula to fill out a table of solution values in a bottom-up manner, from smallest subproblem to largest.
- The formula also suggests a recursive algorithm, but naive recursion can be terribly inefficient, because it solves the same subproblems over and over again.
- Memoization - record the results of previous invocations to avoid repetitions!

A hash table, initially empty, holds values of $K(w)$ indexed by w

<u>function knapsack</u>(w)
if w is in hash table: return $K(w)$
$K(w) = \max\{\texttt{knapsack}(w - w_i) + v_i : w_i \leq w\}$
insert $K(w)$ into hash table, with key w
return $K(w)$