# Lecture 05
# Dynamic programming

Spring 2023

Zhihua Jiang

# Dynamic Programming I:

# Memoization, Fibonacci, Shortest Paths, Guessing

- Memoization and subproblems
- Examples
  - Fibonacci
  - Shortest Paths
- Guessing & DAG View

# Dynamic Programming (DP)

Big idea, hard, yet simple

- Powerful algorithmic design technique
- Large class of seemingly exponential problems have a polynomial solution ("only") via DP
- Particularly for optimization problems (min / max) (e.g., shortest paths)

* DP $\approx$ "controlled brute force"

* DP $\approx$ recursion + re-use

## History

Richard E. Bellman (1920-1984)

Richard Bellman received the IEEE Medal of Honor, 1979. "Bellman ... explained that he invented the name 'dynamid programming' to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who 'had a pathological fear and hatred of the term, research'. He settled on the term 'dynamic programming' because it would be difficult to give a 'pejorative meaning' and because 'it was something not even a Congressman could object to' " [John Rust 2006]

# Fibonacci Numbers

$$F_1 = F_2 = 1; \quad F_n = F_{n-1} + F_{n-2}$$

Goal: compute $F_n$

### Naive Algorithm

follow recursive definition

$\underline{\text{fib}}(n)$:

if $n \leq 2$: return $f = 1$

else: return $f = \text{fib}(n-1) + \text{fib}(n-2)$

$\implies T(n) = T(n-1) + T(n-2) + O(1)$

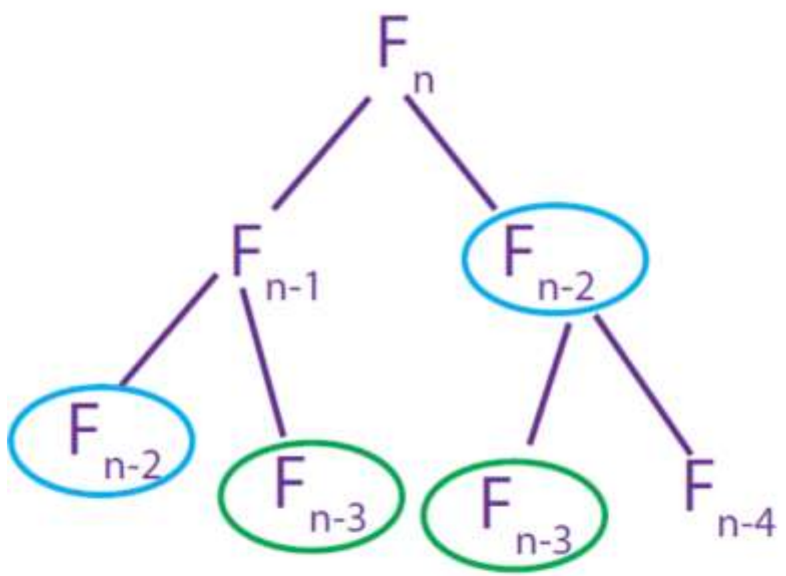
$\geq 2T(n-2) + O(1) \geq 2^{n/2}$

<u>EXPONENTIAL — BAD!</u>

## Memoized DP Algorithm

Remember, remember

memo = { }
fib($n$):
    if $n$ in memo: return memo[$n$]
    else: if $n \leq 2$ : $f = 1$
        else: $f = \text{fib}(n-1) + \text{fib}(n-2)$
        memo[$n$] $= f$
        return $f$



- $\implies \text{fib}(k)$ only recurses <u>first</u> time called, $\forall k$
- $\implies$ only $n$ nonmemoized calls: $k = n, n-1, \ldots, 1$
- memoized calls free ($\Theta(1)$ time)
- $\implies \Theta(1)$ time per call (ignoring recursion)

POLYNOMIAL — GOOD!

* DP $\approx$ recursion + memoization

- memoize (remember) & re-use solutions to subproblems that help solve problem
  - in Fibonacci, subproblems are $F_1, F_2, \ldots, F_n$

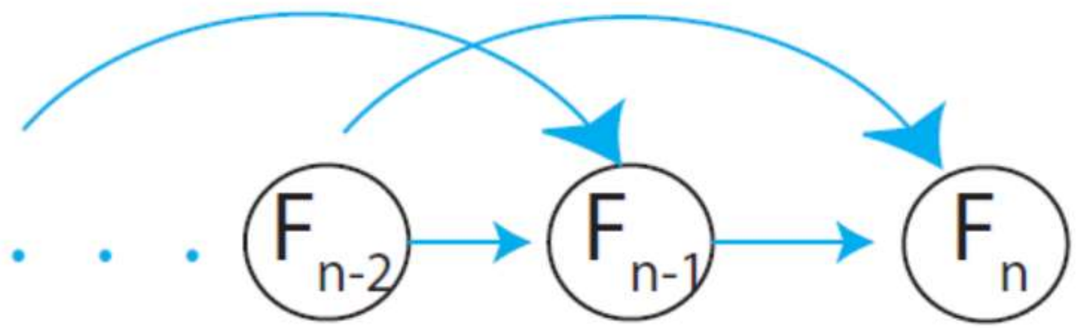
* $\implies$ time = # of subproblems · time/subproblem

- Fibonacci: $\boxed{\text{\# of subproblems is } n\text{, and time/subproblem is } \Theta(1)} = \Theta(n)$ (ignore recursion!).

## Bottom-up DP Algorithm

fib = {} $\quad\quad\quad\quad$ $\Theta(n)$
for $k$ in [1, 2, . . . , $n$]:
  if $k \leq 2$: $f = 1$ $\quad\quad\quad\quad$ $\Theta(1)$
  else: $f = \text{fib}[k-1] + \text{fib}[k-2]$
  fib[$k$] = $f$
return fib[$n$]

- exactly the same computation as memoized DP (recursion "unrolled")
- in general: topological sort of subproblem dependency DAG




- practically faster: no recursion
- analysis more obvious
- can save space: just remember last 2 fibs $\Longrightarrow$ $\Theta(1)$

[Sidenote: There is also an $O(\lg n)$-time algorithm for Fibonacci, via different techniques]

Is there a faster way to compute the $n$th Fibonacci number than by `fib2` (page 13)? One idea involves *matrices*.

We start by writing the equations $F_1 = F_1$ and $F_2 = F_0 + F_1$ in matrix notation:

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

Similarly,

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

and in general

$$\boxed{\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.}$$

So, in order to compute $F_n$, it suffices to raise this $2 \times 2$ matrix, call it $X$, to the $n$th power.

(a) Show that two $2 \times 2$ matrices can be multiplied using 4 additions and 8 multiplications.

But how many $\boxed{\text{matrix multiplications}}$ does it take to compute $X^n$?

(b) Show that $O(\log n)$ matrix multiplications suffice for computing $X^n$. (*Hint:* Think about computing $X^8$.)

a) For any $2 \times 2$ matrices $X$ and $Y$:

$$XY = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix} = \begin{pmatrix} x_{11}y_{11} + x_{12}y_{21} & x_{11}y_{12} + x_{12}y_{22} \\ x_{21}y_{11} + x_{22}y_{21} & x_{21}y_{12} + x_{22}y_{22} \end{pmatrix}$$

This shows that every entry of $XY$ is the addition of two products of the entries of the original matrices. Hence every entry can be computed in 2 multiplications and one addition. The whole matrix can be calculated in 8 multiplications and 4 additions.

b) First, consider the case where $n = 2^k$ for some positive integer $k$. To compute, $X^{2^k}$, we can recursively compute $Y = X^{2^{k-1}}$ and then square $Y$ to have $Y^2 = X^{2^k}$ Unfolding the recursion, this can be seen as repeatedly squaring $X$ to obtain $X^2, X^4, \cdots, X^{2^k} = X^n$. At every squaring, we are doubling the exponent of $X$, so that it must take $k = \log n$ matrix multiplications to produce $X^n$. This method can be easily generalized to numbers that are not powers of 2, using the following recursion:

$$X^n = \begin{cases} (X^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is even} \\ X \cdot (X^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is odd} \end{cases}$$

The algorithm still requires $O(\log n)$ matrix multiplications, of which $\log n$ are squares and at most $\log n$ are multiplications by $X$.

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 $y_1, \ldots, y_j$.

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

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

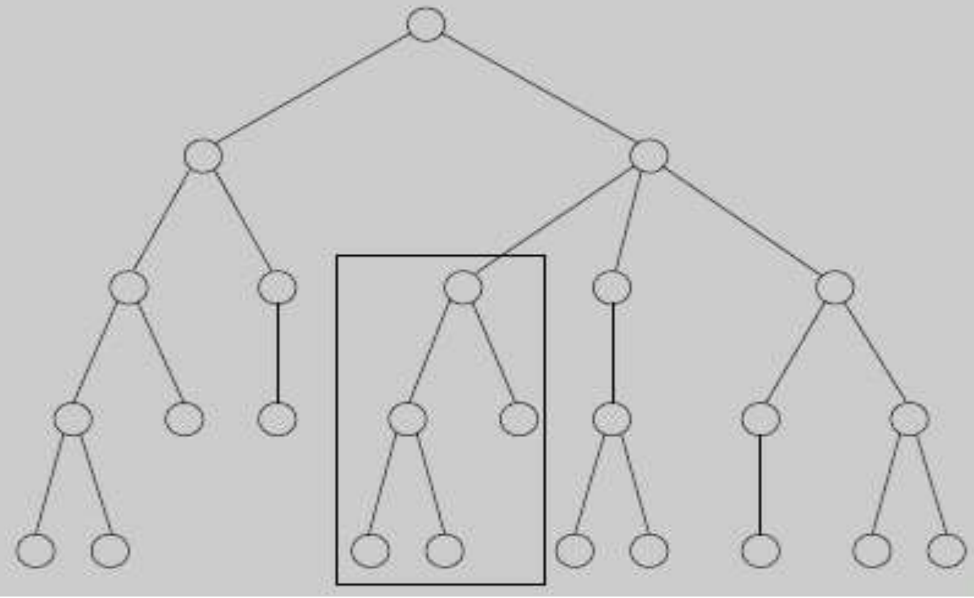
The number of subproblems is $O(mn)$.

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 $O(n^2)$.

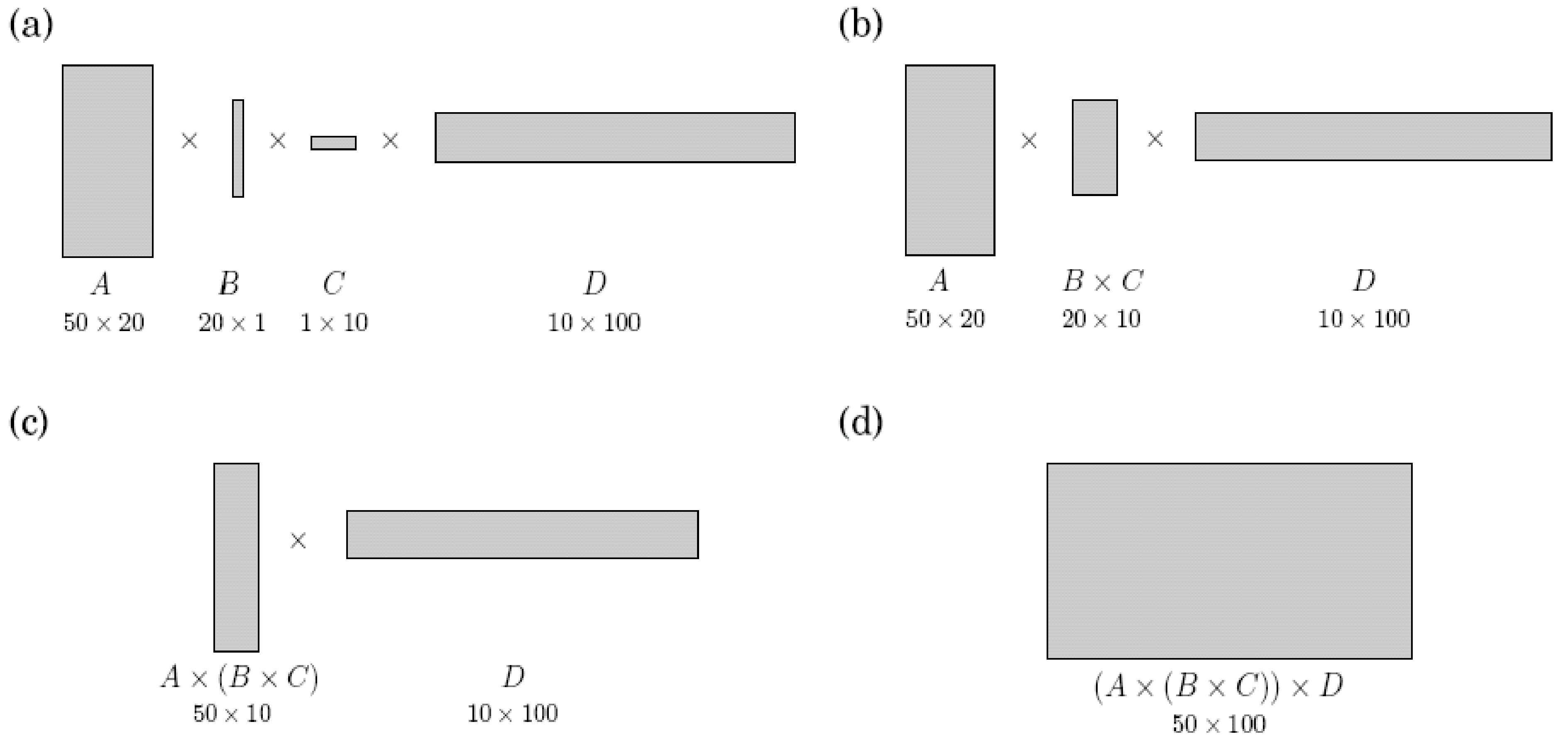
iv. The input is a rooted tree. A subproblem is a rooted subtree.



# Chain matrix multiplication

- Problem: multiply four matrices

**Figure 6.6** $A \times B \times C \times D = (A \times (B \times C)) \times D$.



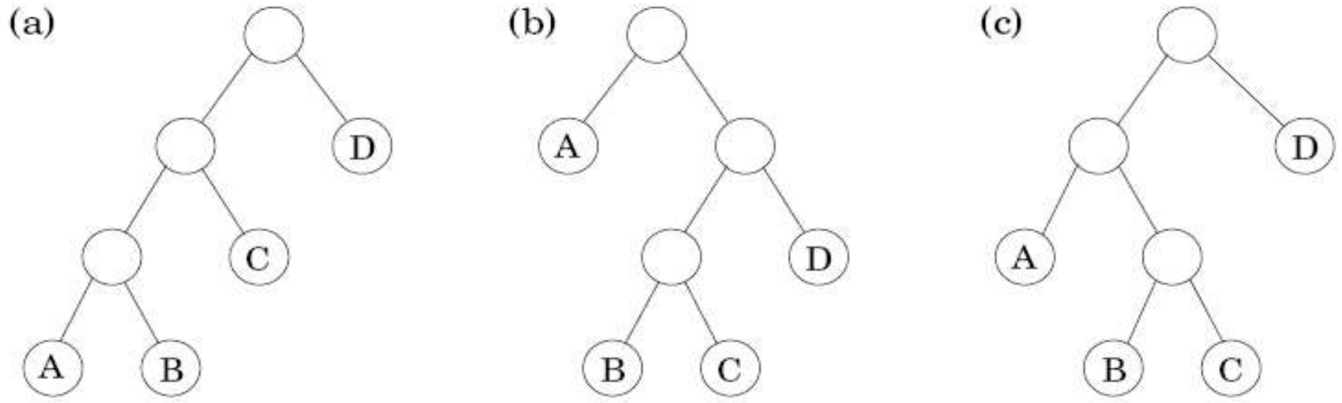
# Chain matrix multiplication

- Matrix multiplication is associative.
- Compute the product of four matrices in many different ways, depending on how parenthesize it.
- Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes $mnp$ number multiplications.
- The order of multiplications makes a big difference in the final running time.

| Parenthesization | Cost computation | Cost |
|---|---|---|
| $A \times ((B \times C) \times D)$ | $20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$ | $120,200$ |
| $(A \times (B \times C)) \times D$ | $20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$ | $60,200$ |
| $(A \times B) \times (C \times D)$ | $50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$ | $7,000$ |

# 6.5 Chain matrix multiplication

- How to determine the optimal order for $A_1 \times A_2 \times \ldots \times A_n$, where $A_i$ with dimension $m_{i-1} \times m_i$?
- A particular parenthesization = a binary tree
  - Leaf: matrix
  - Root: product
  - Interior node: intermediate product
  - The number of binary trees is exponential in $n$ ($n$ leaves)



# Chain matrix multiplication

- The optimal substructure property: for a tree to be optimal, its subtree must also be optimal.
- Subproblem corresponding to a subtree: product of the form $A_i \times A_{i+1} \times \ldots \times A_j$
- For $1 \leq i \leq j \leq n$, the optimal value array

  $C(i,j)$ = minimum cost of multiplying $A_i \times A_{i+1} \times \ldots \times A_j$
- The splitting point $k$: split the product into two pieces, of the form $A_i \times \ldots \times A_k$ and $A_{k+1} \times \ldots \times A_j$
- The total cost is the cost of these two partial products plus the cost of combining them

$$C(i,j) = \min_{i \leq k < j} \{C\{i,k\} + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

How to determine the optimal order for $A_1 \times A_2 \times \ldots \times A_n$, where $A_i$ with dimension $m_{i-1} \times m_i$?

- A dynamic programming algorithm:

for $i = 1$ to $n$: $\quad C(i,i) = 0$
for $s = 1$ to $n-1$:
  for $i = 1$ to $n-s$:
    $j = i + s$
    $C(i,j) = \min\{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j : i \le k < j\}$
return $C(1,n)$

# Edit distance

- The distance between two strings: the extent to which they can be aligned.
- Example: SNOWY vs. SUNNY

| | | | | | |
|---|---|---|---|---|---|
| S | — | N | O | W | Y |
| S | U | N | N | — | Y |

Cost: 3

| | | | | | | |
|---|---|---|---|---|---|---|
| — | S | N | O | W | — | Y |
| S | U | N | — | — | N | Y |

Cost: 5

- *<u>Cost</u>* of an alignment: the number of columns in which the letters differ.
- *<u>Edit distance</u>*: the cost of their best possible alignment, i.e., the minimum number of edits – insertions, deletions, and substitutions – needed to transform the first string into the second.

# Edit distance

- What are the subproblems?
  - Property: there is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to “smaller” subproblems, i.e., subproblems that appear earlier in the ordering.
  - <u>***Subproblem E(i,j):***</u> the edit distance between some prefix of the first string $x[1..i]$ and some prefix of the second $y[1..j]$

The subproblem $E(7, 5)$.

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

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

# Edit distance

- Smaller problems for $E(i,j)$: $E(i\text{-}1,j)$, $E(i,j\text{-}1)$, $E(i\text{-}1,j\text{-}1)$

$$\begin{array}{c} x[i] \\ - \end{array} \quad \text{or} \quad \begin{array}{c} - \\ y[j] \end{array} \quad \text{or} \quad \begin{array}{c} x[i] \\ y[j] \end{array}$$

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

$$\mathit{diff}(i,j) = \begin{cases} 0, \text{if} \quad x[i] = y[j] \\ 1, \text{otherwise} \end{cases}$$

Try all and
Pick best!

$$E(4,3) = \min\{1 + E(3,3),\ 1 + E(4,2),\ 1 + E(3,2)\}.$$

$$\begin{array}{c} \texttt{O} \\ - \end{array} \quad \texttt{or} \quad \begin{array}{c} - \\ \texttt{L} \end{array} \quad \texttt{or} \quad \begin{array}{c} \texttt{O} \\ \texttt{L} \end{array}$$

# Edit distance

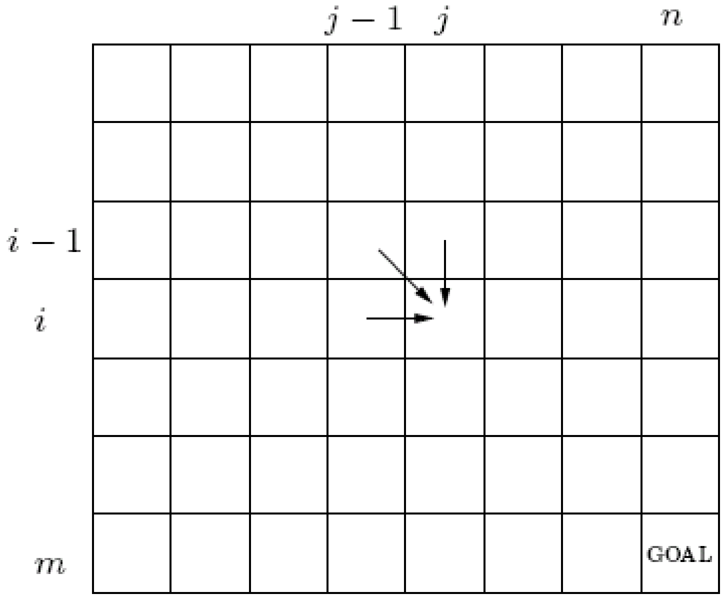
- A dynamic programming solution

```
for i = 0,1,2,...,m:
    E(i,0) = i
for j = 1,2,...,n:
    E(0,j) = j
for i = 1,2,...,m:
    for j = 1,2,...,n:
        E(i,j) = min{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+diff(i,j)}
return E(m,n)
```

- Fill a table row by row, and left to right within each row
- The time complexity: $O(mn)$

# Edit distance

(a)




(b)

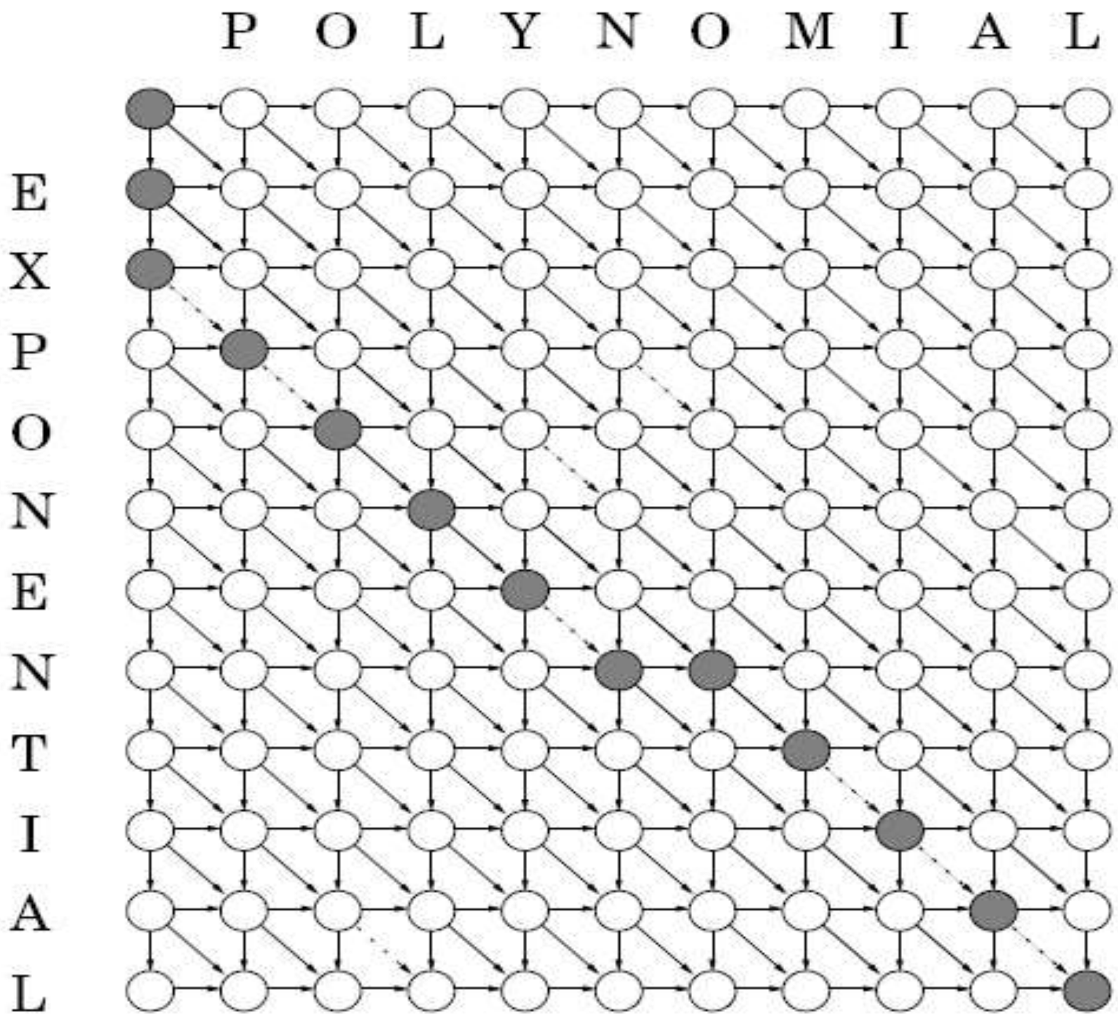
| | | P | O | L | Y | N | O | M | I | A | L |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| E | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| X | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| P | 3 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| O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

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

$$\mathit{diff}(i,j) = \begin{cases} 0, \text{if} \quad x[i] = y[j] \\ 1, \text{otherwise} \end{cases}$$

# Edit distance

- Edits
  - Move down → deletion
  - Move right → insertion
  - Diagonal move → match or substitution



# Knapsack

- Problem description: a total weight $W$, $n$ items to pick, of weight $w_1,\ldots, w_n$ and value $v_1,\ldots, v_n$ , what is the <u>most valuable </u>combination of items which can <u>fit into </u>the bag?

- Example: $W$=10
  - Unlimited quantities of each item:
    optimal solution \$48
    48 = 1 × #1+2 ×#4
  - One of each item:
    optimal solution \$46
    46= 1 × #1+1 ×#3

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

# Knapsack

- Knapsack with repetition
  - Subproblem: smaller knapsack capacities $w \leq W$

  - $K(w)$=maximum value achievable with a knapsack of capacity $w$

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

  - A dynamic programming solution ($W$ and all $w$ need to be integers):

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

  - The time complexity: $O(nW)$

# 6.4 Knapsack

- Knapsack without repetition
  - Subproblem: smaller knapsack capacities $w \leq W$ and fewer items (items 1,2,…,$j$, for $j \leq n$)
  - $K(w, j)$=maximum value achievable with a knapsack of capacity $w$ and items 1,…,$j$

$$K(w,j) = \max\{K(w - w_j, j-1) + v_j, K(w, j-1)\}.$$

  - A dynamic programming solution:

```
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 time complexity: $O(nW)$

for $w = 1$ to $W$:

$$K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\}$$

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

Max{K(4)+30,K(7)+14,K(6)+16,K(8)+9}

Max{K(3)+30,K(6)+14,K(5)+16,K(7)+9}

Max{K(2)+30,K(5)+14,K(4)+16,K(6)+9}

| $w$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $K(w)$ | 0 | 9 | 14 | 18 | 23 | 30 | 32 | 39 | 44 | 48 |

Max{K(0)+9}

Max{K(0)+14,K(1)+9}

Max{K(1)+14,K(0)+16,K(2)+9}

Max{K(2)+14,K(1)+16,K(3)+9}

Max{K(0)+30,K(3)+14,K(2)+16,K(4)+9}

Max{K(1)+30,K(4)+14,K(3)+16,K(5)+9}

```
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}
```

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

| j \ w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 30 | 30 | 30 | 30 | 30 |
| 2 | 0 | 0 | 0 | 14 | 14 | 14 | 30 | 30 | 30 | 44 | 44 |
| 3 | 0 | 0 | 0 | 14 | 16 | 16 | 30 | 30 | 30 | 44 | 46 |
| 4 | 0 | 0 | 9 | 14 | 16 | 23 | 30 | 30 | 39 | 44 | 46 |

# Independent sets in trees

- Dependent set: subset of nodes $S \subset V$, and there are no edges between them
- Finding the largest independent set in a graph is intractable
- However, it can be solved in linear time when the graph is a tree, using dynamic programming
- Algorithm:
  - Start by rooting the tree at any node $r$. Each node defines a subtree.
  - The goal is $I(r)$:

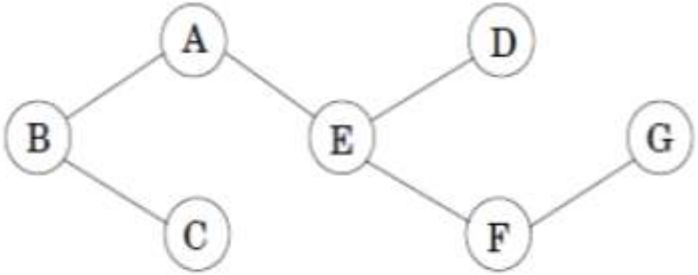
    $I(u)$ = size of largest independent set of subtree hanging from $u$
  - If know $I(w)$ for all descendants $w$ of $u$, then compute $I(u)$:

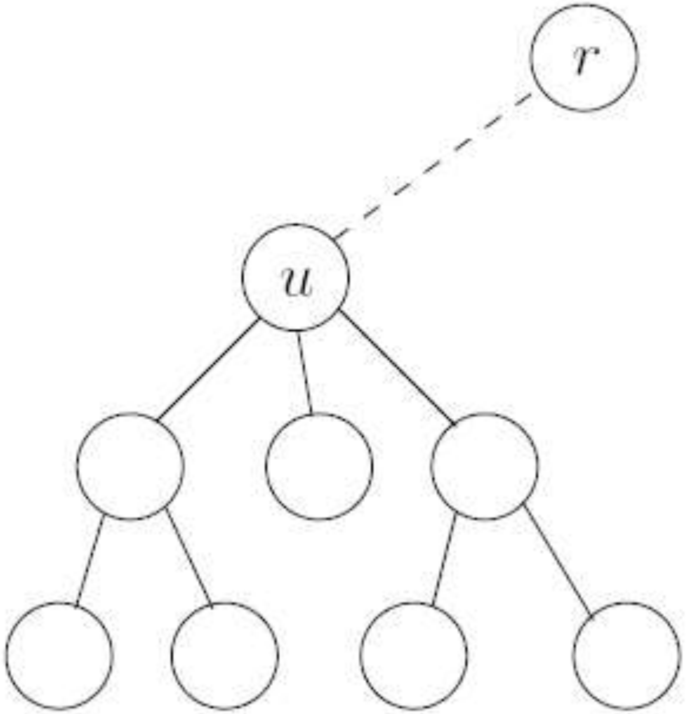
$$I(u) = \max\{1 + \sum_{grandchild} I(gc), \quad \sum_{child} I(c)\}$$

# Independent sets in trees

- The number of subproblems: $O(|V|)$
- The running time: $O(|V|+|E|)$

$$I(u) = \max\{1 + \sum_{grandchild} I(gc), \quad \sum_{child} I(c)\}$$




**I(G)=1**
**I(D)=1**
**$I(F)=\max\{\underline{1},1\}=1$**
**$I(E)=\max\{\underline{1+1},1+1\}=2$**
**I(C)=1**
**$I(A)=\max\{\underline{1+2},2\}=3$**
**$I(B)=\max\{1+2,\underline{3+1}\}=4$**