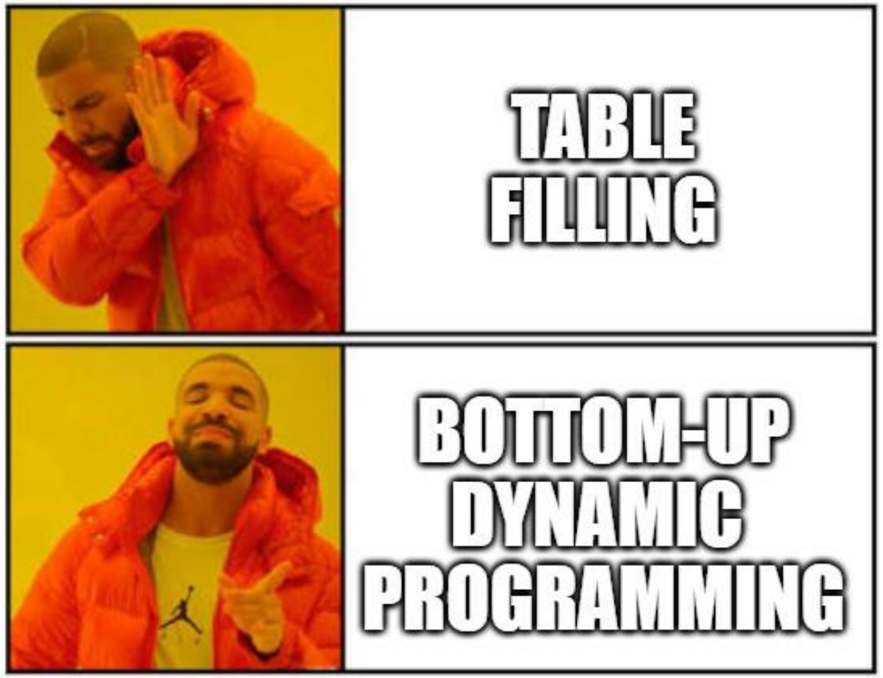
D3: Dynamic Programming



Sec 101: MW 3:00-4:00pm DOW 1018
IA: Eric Khiu

Agenda

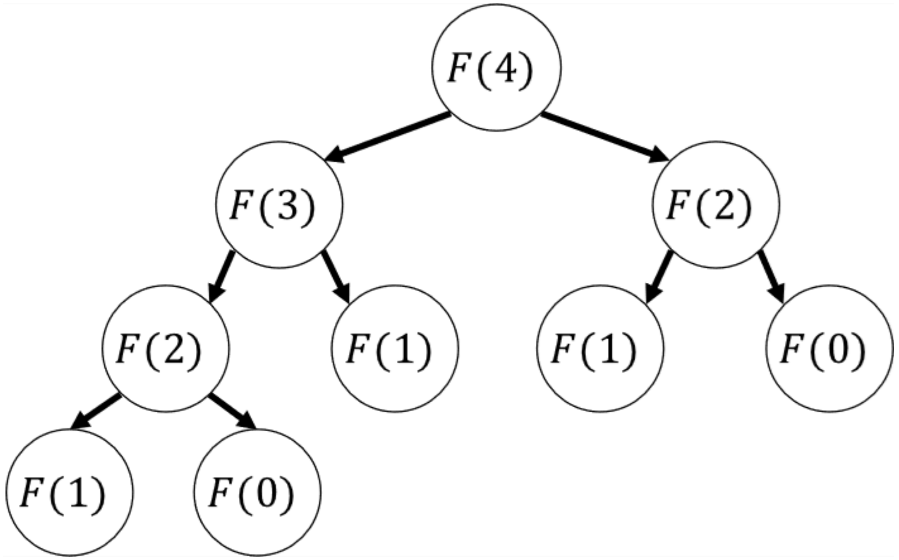
- Dynamic Programming Intro
- DP Implementations
- Subproblems and Dimensionality
- DP Recurrence Relations
- Reconstructing Solution

Dynamic Programming Intro

Notes

Dynamic Programming: Big Idea

- In D&C, we divide a problem into a smaller versions of the same problem
- However, for some problems, this recursive subdivision may result in encountering many instances of the exact same problem
- Wouldn’t it be nice if we remember our solution of duplicated problems so that we don’t have to re-solve them?
- Classic debate: Memory-runtime tradeoff
- In DP, we trade memory for runtime



Divide and Conquer vs DP

Divide and Conquer	Dynamic Programming
Divide original problem to smaller version(s) of the same problem	
Non-overlapping subproblems	Overlapping subproblems
Subproblems usually scale down by a constant: $T(n) \to T\left(\frac{n}{2}\right) \to T\left(\frac{n}{4}\right) \to \cdots$	Subproblems don't usually scale down: $T(n) \to T(n-1) \to T(n-2) \to \cdots$
Optimal substructure: The solution is correct for this (scaled-down) portion	Optimal substructure: The solution is correct up to this point
Always top-town	Top-down, memoization, bottom-up
Often less time efficient	Often more time efficient- especially with bottom-up

Discuss: Why is MergeSort a D&C algorithm rather than a DP algorithm?

No overlapping subproblems- once a subarray is sorted we never have to sort it again

TL; DPA

- We went over the big idea of dynamic programming: trading memory for runtime by remembering answer to overlapping subproblems
- We compare and contrast D&C vs DP

DP Implementations

Notes

Top-down Recursion

- Suppose we want to compute the n-th number in the Fibonacci sequence

$$F(n) = \begin{cases} 1 & \text{if } n \le 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

- Top-down recursion algorithm
  (Implement as in recurrence relation):

$\textsc{Fib}(n)$:

 if $n \le 1$ **then return** 1

 return $\textsc{Fib}(n-1)$ + $\textsc{Fib}(n-2)$

Visualizer

- Easy to translate from recurrence relation
- No additional data structures necessary

- A lot of recursive calls- may not be time-efficient
- Correctness proof is usually harder
  - Not as smooth as bottom up- imagine proving by induction $P(k) \Rightarrow P(k+1)$, but we can't do that with recursive top-down
- Additional concern on segmentation fault

$$F(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

Top-down Memoization

- Same as before, but include a memo recording results of previous recursive calls
- Top-down memoization algorithm:

$memo \leftarrow$ an empty table

FIB(n):

    if $n \leq 1$ **then return 1**

    if $n \notin memo$ **then**

        $memo[n] \leftarrow$ FIB($n-1$) + FIB($n-2$)

    return $memo[n]$

- Usually better time-complexity than top-down recursion
- While harder than recursion, the logic is often more intuitive than bottom-up

- May still be slower than bottom-up
- Correctness proof may still be harder than bottom-up
- Additional concern on segmentation fault

$$F(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

Bottom-up (Tabulation)

- Build the table without recursion
- Iterate over previous results to fill the cells
- Bottom-up algorithm:

FIB(n):

$table \leftarrow$ an empty table
$table[0] \leftarrow 1$
$table[1] \leftarrow 1$

for $i = 2, \ldots, n$ **do**
  $table[i] \leftarrow table[i-1] + table[i-2]$

return $table[n]$

- Almost always fastest in practice
- Segmentation fault is less likely

- Less intuitive
- Common mistake 1: wrong direction for for-loop
- Common mistake 2: wrong initialization

TL; DPA

- We compared and contrasted the three methods for implementing a DP algorithm
- For this class, we expect you to know how to implement DP bottom-up

Doul P.

Lingering question:
Does a bottom-up solution always exists?

Yes! See back matter.

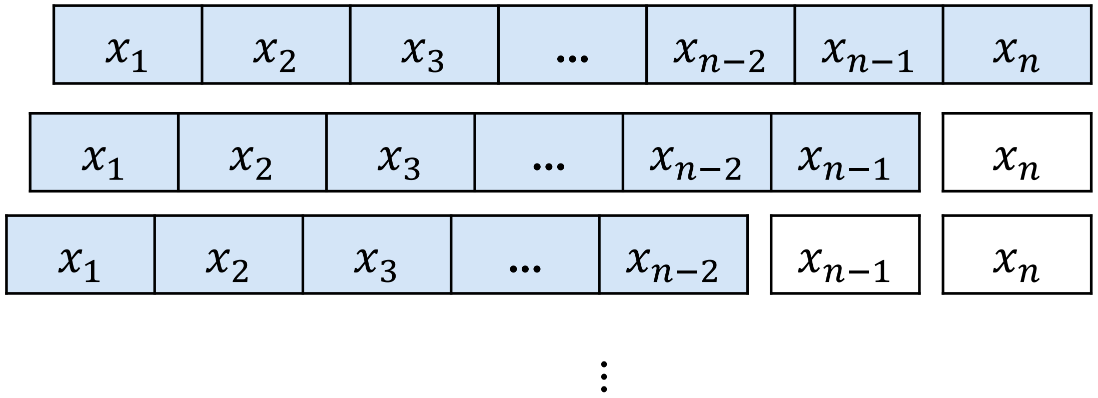
Bottom-up DP Cookbook

- Write recurrence
- Size of table (Dimensions? Range of each dimensions?)
- To fill in cell, which other cells do I look at?
- Which cell(s) contain the final answer?
- Reconstructing solution: Follow arrows from final answer to base case

Subproblems and Dimensionality

Subproblems and Dimensionality

- Recall the idea of DP is to reduce the original problem into smaller subproblems
- Therefore, it is important to first determine
  - In how many directions are we reducing the problem (dimension)
  - In what direction(s) are we reducing the problem (will discuss later in recurrence relation construction)
- Here's a common 1-dimensional subproblem reduction:



Discuss: How many subproblems do we have to solve in total asymptotically?

$O(n)$

Example: Longest increasing subsequence

Common 2-dimensional Subproblems

- Here's a common 2-dimensional subproblem reduction

	x_1	x_2	x_3	…	x_{n-2}	x_{n-1}	x_n
y_1							
y_2							
y_3							
⋮							
y_{m-2}							
y_{m-1}							
y_m							

	x_1	x_2	x_3	…	x_{n-2}	x_{n-1}
y_1						
y_2						
y_3						
⋮						
y_{m-2}						
y_{m-1}						

x_n

y_m						

Example: Longest common subsequence, 0-1 knapsack

TL; DPA

- We discussed the step 0 in tackling a DP problem- identifying the dimensionality
- This is a small but crucial step in setting up a DP table
- We could think of dimension as the number of variables we need to keep track of

DP Recurrence Relations

DP Recurrence

- Often the trickiest part of DP problems
- My recipe: three steps
- **Step 1: Define the subject of recurrence** (in English!)
  - For example, in LIS we have “LIS(i) = the longest increasing subsequence ending at A[i]”
- **Step 2: Identify the base case(s)**
  - At this point, we should have a vague idea on the dimensionality and the direction(s) we want to reduce the problem in
  - Base case = in what scenario we can’t reduce the problem further
- **Step 3: Construct (optimal) sub-solution**
  - [Sub] How and when to reduce into smaller version of the same problem?
  - [Solution] How to combine the result so that the overall result is correct? (Typically involves choosing something)
  - [Optimal] Max/ Min? Objective function? (Only for optimization problem)

Smallest Subarray Product

Given an array $A[1, \ldots, n]$ *positive real numbers*, we are interested in finding the smallest product of any subarray (selected elements must be contiguous) of A. For example:

A = [2, 0.5, 4, 0.1]

Devise a recurrence relation for a dynamic programming algorithm for this problem.

Poll: Which recurrence relation is more suited for this problem and why?
A. $S(i)$ = smallest product of subarray *within* $A[1, .., i]$
B. $S(i)$ = smallest product of subarray *ending at* $A[i]$

Why doesn't option A work?

- Consider the example from earlier: $A = [2, 0.5, 6, 0.1]$
- If we let $S(i)$ = smallest product of subarray *within* $A[1, .., i]$, then we have

i	All possible subarrays	$S(i)$
1	$[2]$	2
2	$[2]$, $[0.5]$ $[2, 0.5]$	0.5
3	$[2]$, $[0.5]$, $[6]$ $[2, 0.5]$, $[0.5, 6]$ $[2, 0.5, 6]$	4
4	$[2]$, $[0.5]$, $[6]$, $[0.1]$ $[2, 0.5]$, $[0.5, 6]$, $[6, 0.1]$ $[2, 0.5, 6]$, $[0.5, 6, 0.1]$ $[2, 0.5, 6, 0.1]$	0.3

At i, we only have access to

- $A[i]$
- $S(j)$s for all $j < i$

BAD: We can't reuse previous results $S(j)$s

Smallest Subarray Product

- **Step 1: Subject of recurrence**
  - $S(i)$ = smallest product of subarray *ending at* $A[i]$
- **Step 2: Base case(s)**
  - $i = 1$: Only one element, so $S(i) = A[i]$
- **Step 3: Optimal sub-solution**
  - [Sub] Before solving for $L(i)$, solve for $L(i-1)$
  - [Solution] We have two choices:
    - [1] Multiply the subarray considered at $L(i-1)$ to $A[i]$
    - [2] Don't include the subarray considered at $L(i-1)$, let $A[i]$ as the beginning of a subarray
  - [Optimal] Minimization problem: Choose the smallest between [1] and [2]!

$$S(i) = \min\{S(i-1) \cdot A[i], A[i]\}$$

Smallest Subarray product

- Putting everything together, we have the following recurrence relation

$$S(i) = \begin{cases} A[i] & \text{if } i = 1 \\ \min\{L(i-1) \cdot A[i], A[i]\} & \text{otehrwise} \end{cases}$$

- Before writing the algorithm, which cell contains the final solution (smallest subarray product)?

$$\max_{i:1 \le i \le n} S(i)$$

- Translating recurrence relation to algorithm is often mechanical

```
SSP(A[1, …, n]):
    Initialize empty array DP of size n
    DP[1] ← A[i]
    for i = 2, …, n do
        DP[i] ← min{DP[i − 1] · A[i], A[i]}
    return max_{i:1≤i≤n} DP(i)
```

Runtime: $O(n)$

Longest Palindromic Subsequence

A *palindrome* is a string which is the same forwards and backwards (e.g., “racecar”).

Given a string S, write a recurrence relation to determine the *length* of the longest subsequence of S (not necessarily a substring) that is a palindrome.

For example, LPS(“abca”) = 3 (“aba” or “aca”)

Discuss: Why can’t we solve using 1-dimensional DP?

Longest Palindromic Subsequence

- **Step 0: Dimensionality**
  - 2-dimensional (one for start and another for end)
- **Step 1: Subject of recurrence**
  - $LPS(i, j)$ = length of the longest palindromic subsequence in $S[i, \dots, j]$
- **Step 2: Base case(s)**
  - $i = j$ (string of length 1): $LPS(i, j) = 1$
  - $i > j$ (invalid input): $LPS(i, j) = 0$

Longest Palindromic Subsequence

- Step 3: Optimal sub-solution
  - [Sub-solution] From S[i, …, j], how to reduce the problem?
    - Compare S[i] and S[j]
    - If S[i] = S[j], add 2 and recurse into S[i+1, …, j-1]
    - If S[i] ≠ S[j], what options do we have?
      - [1] Recurse into S[i+1, …, j]
      - [2] Recurse into S[I, …, j-1]
  - [Optimal] *Choose* the best option
    - If A[i] = B[j], we don't have to choose: $2 + LPS(i+1, j-1)$
    - If A[i] ≠ B[j], we have a maximization problem: Use max across options [1] and [2]

$$\max\{LPS(i+1, j), LPS(i, j-1)\}$$

Longest Palindromic Subsequence

- Putting everything together, we have $LPS[i,j]$ = length of the longest palindromic subsequence of $S[i,\ldots,j]$, i.e.,

$$LPS[i,j] = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ 2 + 2 + LPS(i+1, j-1) & \text{if } S[i] = S[j] \\ \max\{LPS(i+1,j), LPS(i,j-1)\} & \text{otherwise} \end{cases}$$

- Now consider the bottom-up implementation (suppose we have the LPS table as defined above), which cell contains the final solution (length of LPS in S)?

$$LPS(1,n)$$

Matrix Multiplication

The following is a short example of how to perform matrix multiplication:

If $A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$ and $B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$, then $AB = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{bmatrix}$.

Math recap: What is the number of multiplications in AB in terms of

- A.r = number of rows of matrix A
- A.c = number of columns of matrix A
- B.r = number of rows of matrix B
- B.c = number of columns of matrix B

Ans: A.r · A.c · B.c = A.r · B.r · B.c

Matrix Multiplication

Given a sequence of n matrices to multiply together, write a recurrence to determine the minimum number of element-element multiplications required (you may disregard additions).

As an example, suppose the sequence of matrices to multiply is ABC, where A is 2×3, B is 3×4, and C is 4×5.

Computing $(AB)C$ requires $2 \cdot 3 \cdot 4 + 2 \cdot 4 \cdot 5 = 64$ multiplications. Computing $A(BC)$ requires $3 \cdot 4 \cdot 5 + 2 \cdot 3 \cdot 5 = 90$ multiplications. Therefore, the minimum number of multiplications required is 64.

Hint: Consider multiplying matrices $M_i M_{i+1} \dots M_j$

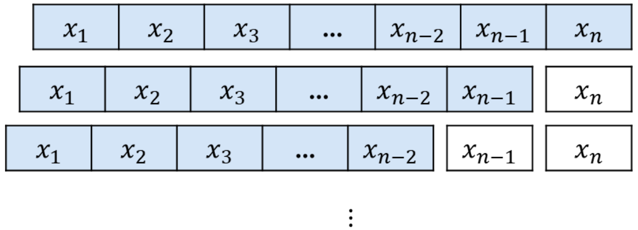
Visualizer

Matrix Multiplication: Dimensionality

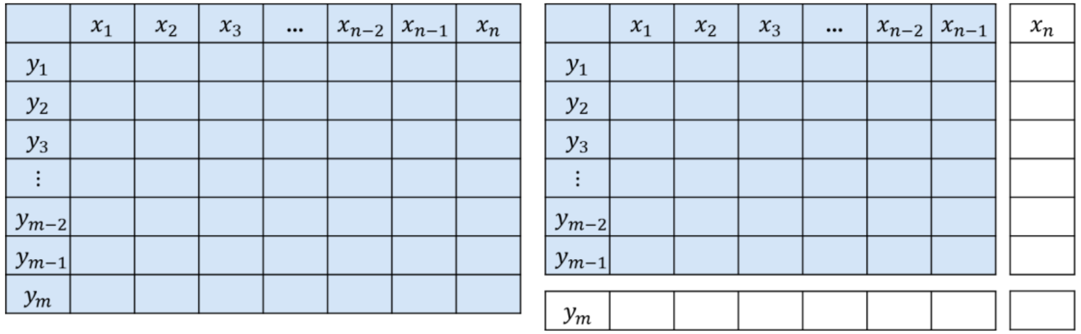
Given a sequence of n matrices to multiply together, write a recurrence to determine the minimum number of element-element multiplications required (you may disregard additions).

Discuss: Is this a 1-dimensional or 2-dimensional problem?

- 1-dimensional doesn't work!
  - Consider multiplying $ABCD$
  - The optimal solution is $(AB)(CD)$
  - But the optimal solution for subproblem ABC is $A(BC)$- how do we combine this result with D?



- Step 0: Dimensionality
  - 2-dimensional (start and end of the matrix sequence)



Matrix Multiplication: Recurrence

- **Step 1: Subject of recurrence**
  - Let $L(i, j)$ = minimum number of multiplications to multiply matrices i through j
- **Step 2: Base cases**
  - $i = j$ (One matrix): $L(i, j) = 0$
- **Step 3: Optimal sub-solution**
  - [Sub-solution] For $M_i M_{i+1} \ldots M_j$, how do we reduce the problem?
    - "Partition" by some k: $(M_i \ldots M_k)(M_{k+1} \ldots M_j) = AB$
    - A is M_i.r × M_k.c; B is M_{k+1}.r × M_j.c = M_k.c × M_j.c
    - Number of multiplications to multiply AB? M_i.r · M_k.c · M_j.c from previous slide
  - [Optimal] *Choose* the best k
    - Minimization problem: Use min across all k's in range $i \leq k < j$
    - Objective function? $L(i, j) = \min_{k: i \leq k < j} \left(L(i, k) + L(k + 1, j) + M[i].r \cdot M[k].c \cdot M[j].c \right)$

Visualizer

DP Recurrence- Takeaway

- **Step 0+1: Dimensionality + Subject of recurrence**
  - Is this a 1D DP problem? 2D? What dimension do we want to do it in?
  - In which/ how many direction(s) do we want to reduce the problem?
- **Step 2: Identify the base case(s)**
  - In what situation(s) we can't reduce the problem further?
  - Is there any special cases?
- **Step 3: Construct (optimal) sub-solution**
  - Sub-solution:
    - [Sub] How to reduce the problem to smaller version of the same problem?
    - [Solution] How to combine the result so that the overall result is correct?
    - If [some condition] is/ isn't satisfied, what options do we have?
  - Optimal: (only for optimization problem)
    - Is it a maximization or minimization problem?
    - What is/ are the variable(s) we're taking max/ min over?
    - What is the objective function to be maximized/ minimized?

DP Reconstructing Solution

Reconstructing Solution

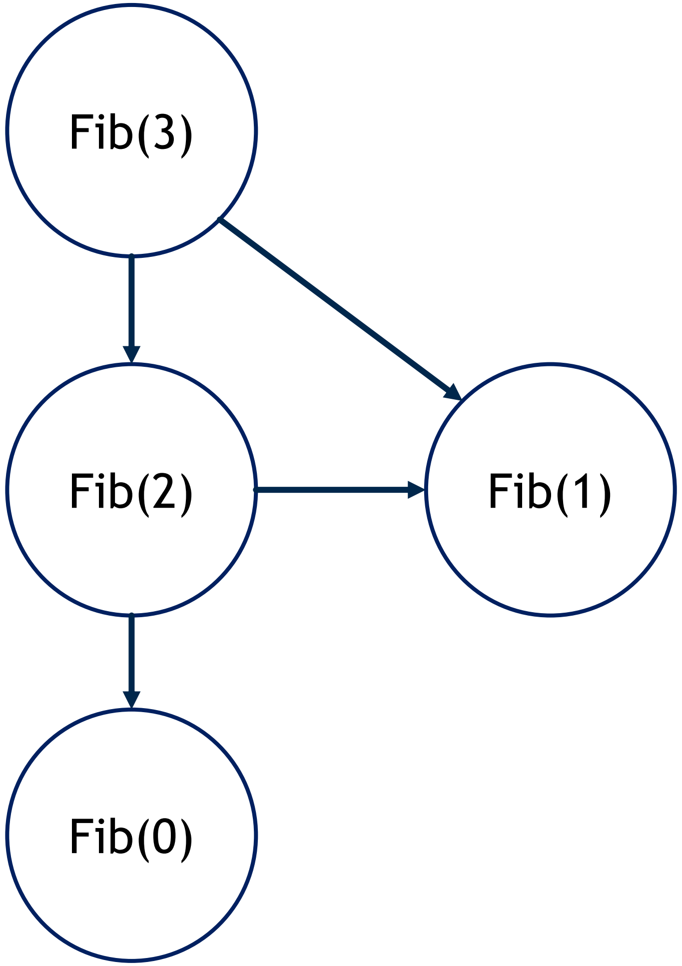
- Identify the cell containing final solution, c_s
- Identify the cell containing the base case
- Track all the cells used to fill c_s, cells used to fill those cells, …, all the way to the base case

- **Remark:** In OOP, it might be easier to represent each cell as an object and store the "where I come from" info

Back Matter

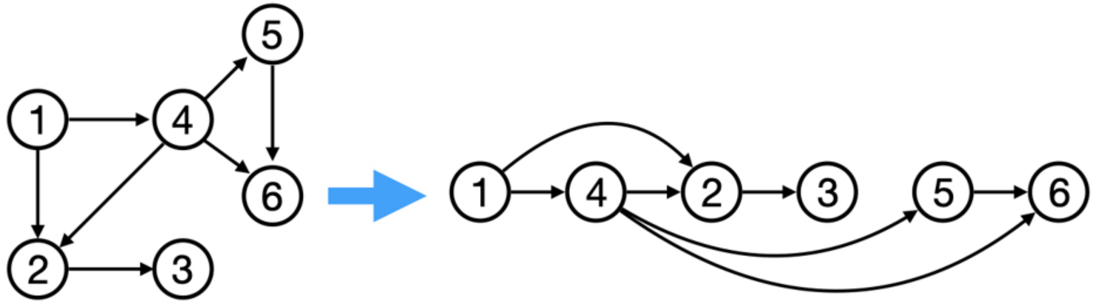
Does Bottom-up Solution Always Exists?

- Yes! And interestingly, we will use Graph Theory to prove!
- First, we claim that a dynamic programming problem can be visualized as a Directed Acyclic Graph (DAG)
  - Each node in the DAG represents a subproblem
  - A directed edge from node A to node B implies that solving subproblem A requires the solution from subproblem B
- **Acyclic:** Key property of DP subproblems: a subproblem will not depend on itself or any larger subproblem



DAG Topological Sort

- **Topological Sort for DAG:** A linear ordering of vertices such that for every directed edge (u, v) from vertex u to vertex v, u comes before v in the ordering
- Here, the order represents the order of subproblems we're trying to solve (Top-down: sequence of recursive calls)
- If a DAG is topologically sortable, each top-down approach correlates to a bottom-up approach by processing nodes in reverse topological order
  - Think about why this is true!
- There are a lot of algorithms to topologically sort a DAG. One of them is the Kahn's algorithm




```
KahnAlg(G = (V, E)):
    initialize i ← 0 and an empty array L[1, …, |V|]
    S ← {u ∈ V: u has no incoming edge}
    while S is not empty do
        S ← S \ {u} for some arbitrary u ∈ S
        i ← i + 1, L[i] ← u
        for each outgoing e = (u, v) ∈ E from u do
            E ← E \ {u}
            if v has no incoming edge then
                S ← S ∪ {v}
    return L
```

Useful Notations

Sum Σ/ product Π

Over an array $A[1,\dots,n]$

$$\sum_{i=1}^{n} A[i] = A[1] + A[2] + \cdots + A[n]$$

$$\prod_{i=1}^{n} A[i] = A[1] \cdot A[2] \cdot \dots \cdot [n]$$

Over a set $S = \{s_1,\dots,s_n\}$

$$\sum_{s \in S} s \qquad \prod_{s} s$$

Conditional

$$\sum_{i:\, i \text{ is even}} A[i] \qquad \prod_{s \in S:\, s>0} s$$

Max/ min

Over an array $A[1,\dots,n]$ $\quad \max\limits_{i:\, 1 \le i \le n} A[i]$

Over a set $S = \{s_1,\dots,s_n\}$ $\quad \min\limits_{s \in S} s$

Sets Operations

Union $\quad \bigcup_{i=1}^{n} S_i = S_1 \cup S_2 \cup \cdots \cup S_n$

Intersection $\quad \bigcap_{i=1}^{n} S_i = S_1 \cap S_2 \cap \cdots \cap S_n$

Set minus $\quad A \setminus B = A \cap \bar{B}$

It is common to write $S \leftarrow S \setminus \{s\}$ for "remove element s from set S"

Big AND ∧ / big OR ∨

Over an array of truth values $T[1,\dots,n]$

$$\bigwedge_{i=1}^{n} T[i] = T[1] \wedge T[2] \wedge \cdots \wedge T[n]$$

$$\bigvee_{i=1}^{n} T[i] = T[1] \vee T[2] \vee \cdots \vee T[n]$$

Conquering the Fear of DP

- **You're Not Alone!** Many of us find dynamic programming challenging at first
- **Understand the Problem:** Regularly practice reading prompts to grasp the problem's requirements
  - Check out geeksforgeeks.org for list of DP problems- getting comfortable reading the prompt makes you confident
- **Visualizers:** Explore visualizers to understand the shape and structure of the DP table
  - Check out Algorithm visualizer
- **Break It Down:** Tackle each component of dynamic programming separately to avoid feeling overwhelmed:
  - Dimensionality
  - Recurrence relation
  - Implementation
  - Reconstructing solution

Memes I couldn’t decide which one to use