香港科技大学(广州)
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

Dynamic Programming

Objective

- Dynamic programing: new technique for solving optimization problems.
- Understand why we use dynamic programming.
- Apply DP on many examples.

What is Dynamic Programing (DP) ?

- First used by Richard Bellman in the 1950s

- Conceived to optimally plan multistage processes

- Usually refers to simplifying a decision by breaking it down into a sequence of decision steps over time

Main Ideas

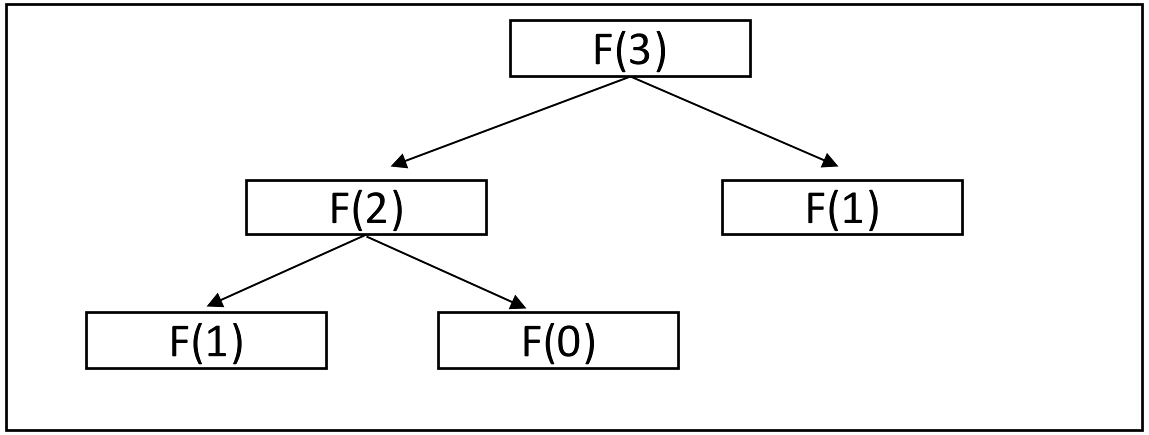
1. Recursion: Divide the problem into sub-problems, so that their solutions can be combined into a solution to the problem.

2. Tabulation of sub-problems: Solve each sub-problem just once and save its solution in a "look-up" table.

Fibonacci Numbers

- Computing the n^{th} Fibonacci number recursively:
  - $F(n) = F(n-1) + F(n-2)$
  - $F(0) = 0$
  - $F(1) = 1$

```
def Fib(n):
   if (n <= 1)
      return 1;
   else
      return Fib(n - 1) + Fib(n - 2);
```



Fibonacci Numbers

- What is the Recurrence relationship?
  - $T(n) = T(n-1) + T(n-2) + 1$
- What is the solution to this?
  - Clearly it is $O(2^n)$, but this is not tight.
  - A lower bound is $\Omega(2^{n/2})$.
  - You should notice that $T(n)$ grows very similarly to $F(n)$, so in fact $T(n) = \Theta(F(n))$.
- Obviously not very good, and we know that there is a better way to solve it!

Fibonacci Numbers

- Computing the n^th^ Fibonacci number using as follow:
  - F(0) = 0
  - F(1) = 1
  - F(2) = 1+0 = 1
  - …
  - F(n-2) =
  - F(n-1) =
  - F(n) = F(n-1) + F(n-2)

0	**1**	**1**	**. . .**	***F*(*n*-2)**	***F*(*n*-1)**	***F*(*n*)**

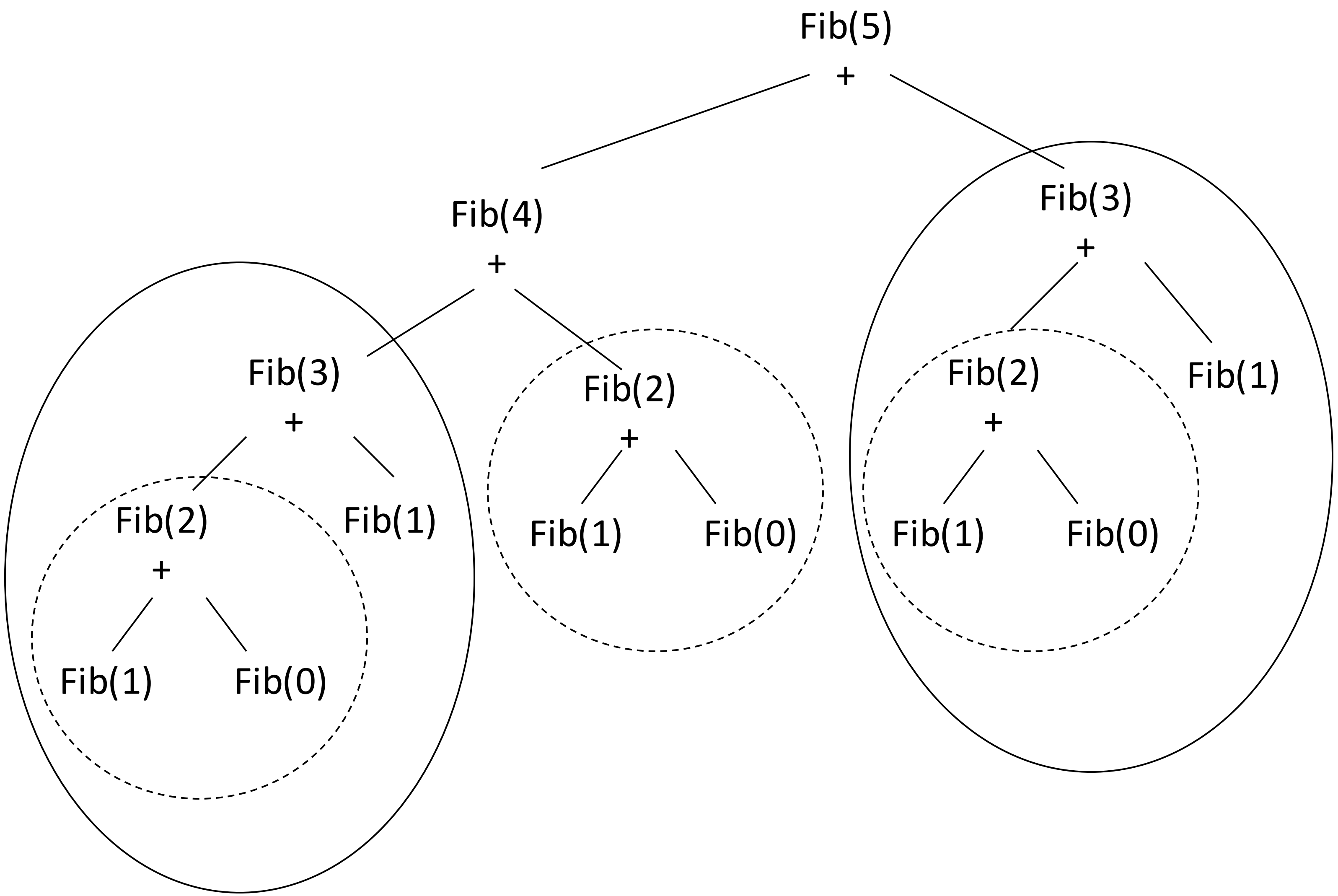
- Efficiency:
  - Time – O(n)
  - Space – O(n) ➔ can be improved to O(1)

Fibonacci Numbers

- The approach is only $\Theta(n)$.

- Why is the naive recursion so inefficient?
  - Recomputes many sub-problems.
    - How many times is F(n-3) computed? Try to draw a solving tree by yourself and answer this question.
    - Does F(n-3) necessarily need to be computed so many times?

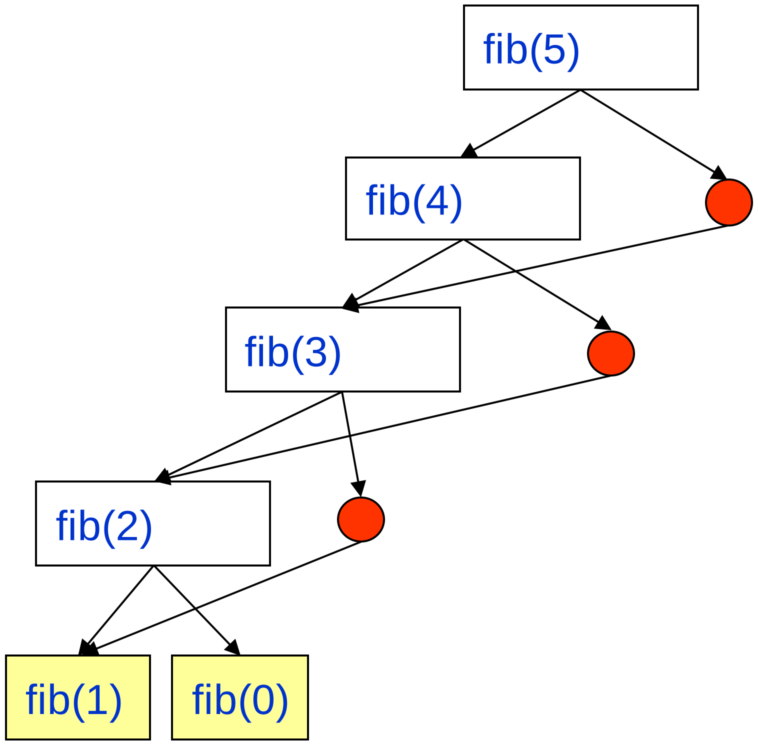
Fibonacci Numbers



Memoization

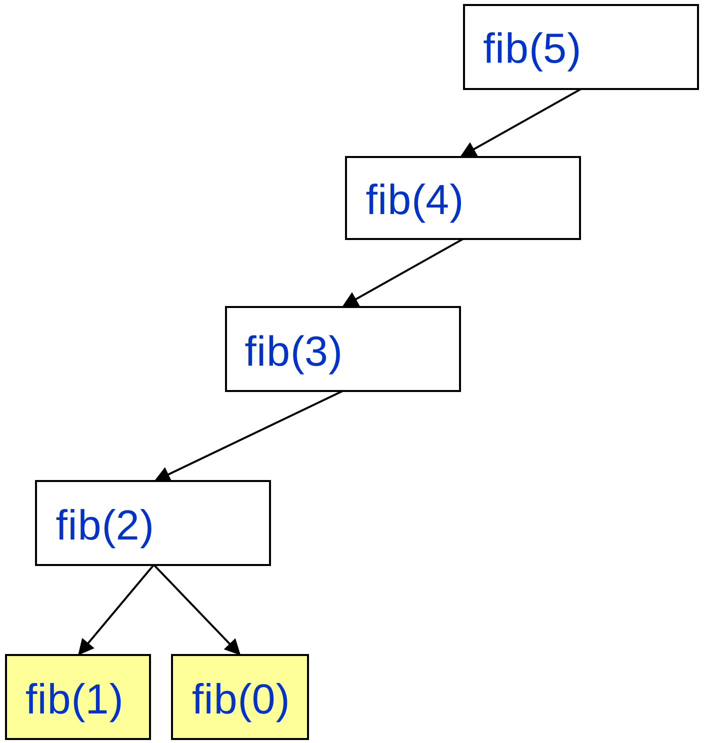
- Easy solution to avoid duplicate computation: ‘memoization’
  - Remember solutions of all the sub-problems
  - Trade space for time

Sub-problem	Opt Solution
fib(4)	3
fib(3)	2
fib(2)	1
fib(1)	1
fib(0)	0

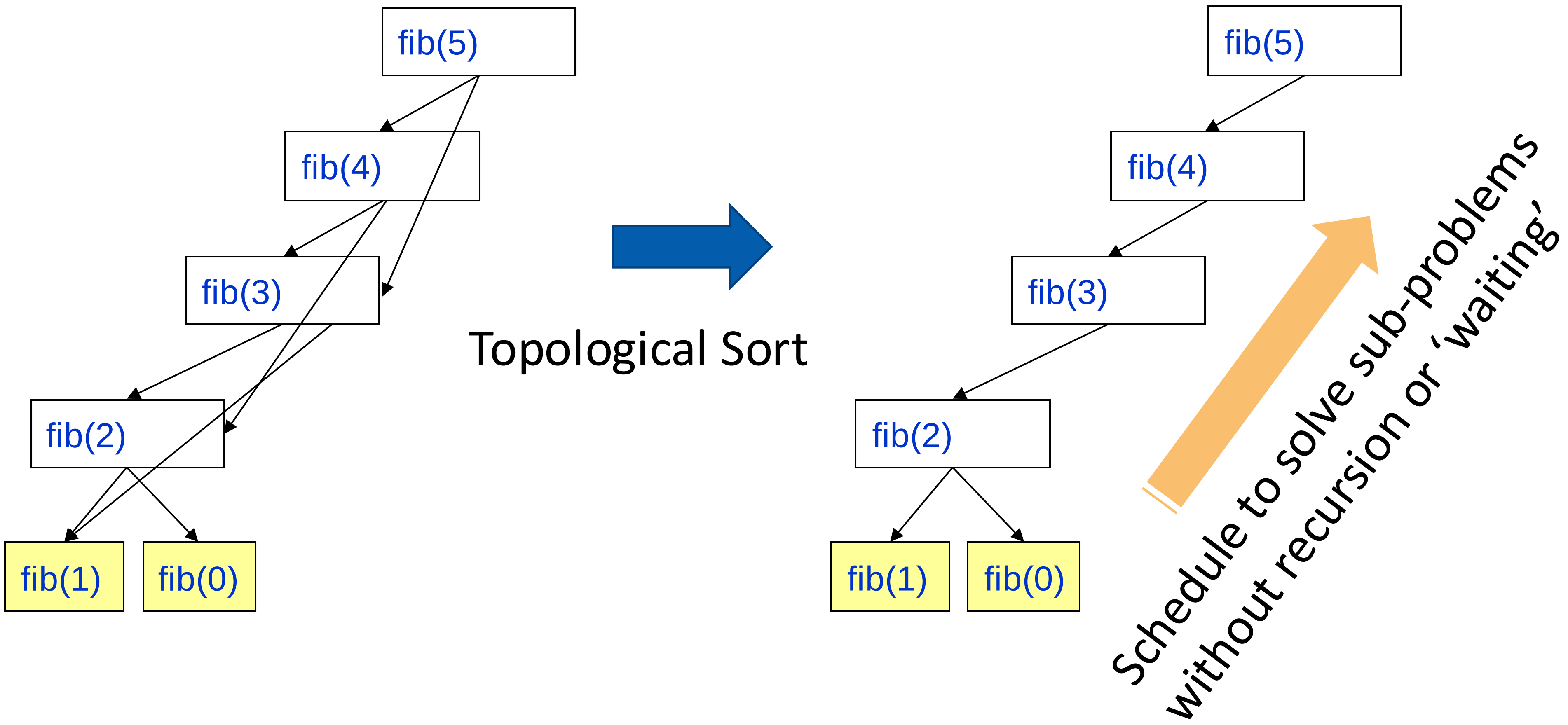


Dynamic Programming

- Ideas
  - Ensure all needed recursive calls are already computed and memorized ➔ a good schedule of computation
  - (Optional) Reused space to store previous recursive call results
  - ➔ Arrive at the same efficient (special) solution for Fib()



Analyze the sub-problems



This process is Dynamic Programming!

Dynamic Programming

- Dynamic Programming is an algorithm design technique for *optimization problems:* often minimizing or maximizing.

- Like divide and conquer, DP solves problems by combining solutions to sub-problems.

- Unlike divide and conquer, sub-problems are not independent.
  - Sub-problems may share sub-sub-problems.

Dynamic Programming

- The term Dynamic Programming comes from Control Theory, not computer science. Programming refers to the use of tables (arrays) to construct a solution.
- In Dynamic Programming, we usually reduce time by increasing the amount of space.
- We solve the problem by solving sub-problems of increasing size and saving each optimal solution in a table (usually).
- The table is then used for finding the optimal solution to larger problems.
- Time is saved since each sub-problem is solved only once.

Two Ways to Think and Implement DP

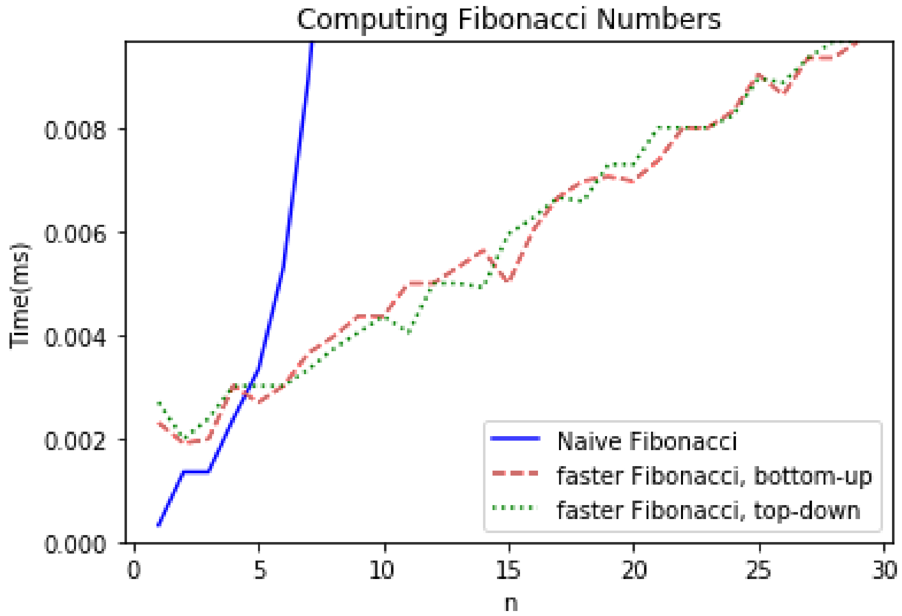
- Top down:
- Think of it like a recursive algorithm.
- To solve the big problem:
  - Recurse to solve smaller problems
    - Those recurse to solve smaller problems
      - etc..
- The difference from divide and conquer:
  - Keep track of what small problems you’ve already solved to prevent re-solving the same problem twice.
  - Aka, “**memoization**”

- Bottom up:
- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
- …
- Then bigger problems
  - fill in F[n-1]
- Then finally solve the real problem.
  - fill in F[n]

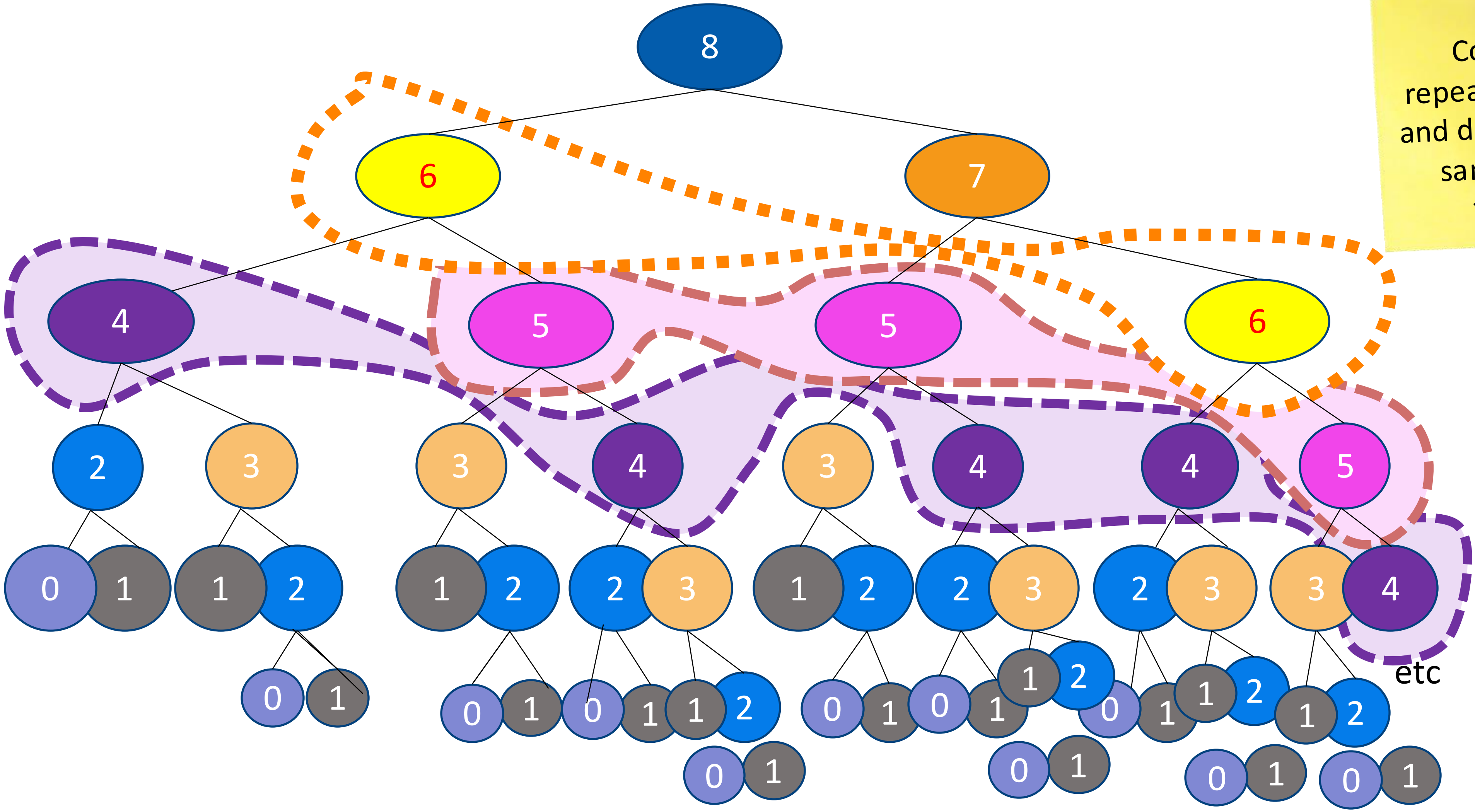
Example of Top-Down Fibonacci

```
• define a global list F = [0,1,None, None, …, None]
• def Fibonacci(n):
    • if F[n] != None:
        • return F[n]
    • else:
        • F[n] = Fibonacci(n-1) + Fibonacci(n-2)
    • return F[n]
```

Memoization: Keeps track (in F) of the stuff you've already done.



Memoization Visualization



Collapse repeated nodes and don't do the same work twice!

The Process of Applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

Dynamic Programming

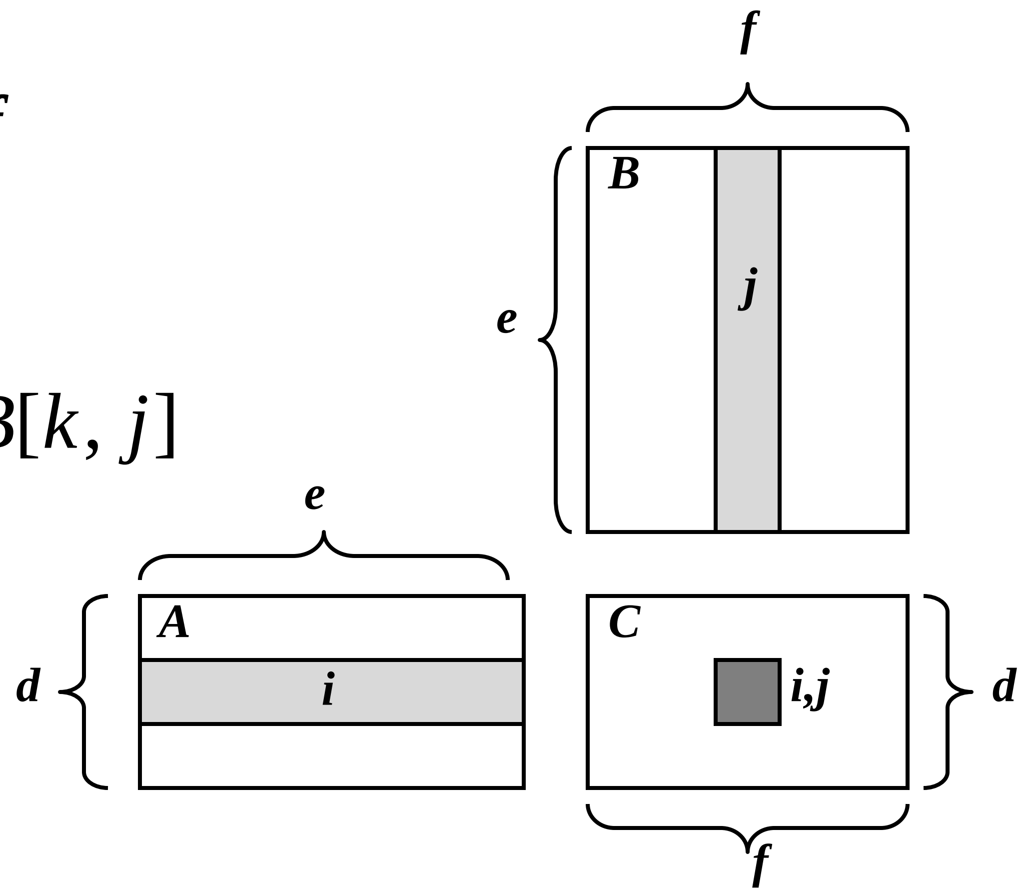
- Underpins many optimization problems, e.g.,
  - Matrix Chaining optimization
  - Longest Common Subsequence
  - 0-1 Knapsack Problem
  - Transitive Closure of a direct graph
  - Shortest path

- Next we will give many example problems to help understand the basic idea of Dynamic Programming.

Matrix Chain-Products

- Review: Matrix Multiplication.
  - $\boldsymbol{C} = \boldsymbol{A} * \boldsymbol{B}$
  - $\boldsymbol{A}$ is $\boldsymbol{d} \times \boldsymbol{e}$ and $\boldsymbol{B}$ is $\boldsymbol{e} \times \boldsymbol{f}$
  - $\boldsymbol{O(d \cdot e \cdot f)}$ time

$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$



Matrix Chain-Products

- **Matrix Chain-Product:**
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- Example
  - B is 3×100
  - C is 100×5
  - D is 5×5
  - (B*C)*D takes 1500 + 75 = 1575 ops
    - (3 x 100 x 5) + (3 x 5 x 5)
  - B*(C*D) takes 1500 + 2500 = 4000 ops

Enumeration Approach

- **Matrix Chain-Product Alg.:**
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best
- Running time:
  - The number of parenthesizations is equal to the number of binary trees with $n-1$ nodes
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

Greedy Approach

- Idea #1: repeatedly select the product that uses the fewest operations.
- Counter-example:
  - A is 101×11
  - B is 11×9
  - C is 9×100
  - D is 100×99
  - Greedy idea #1 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
- The greedy approach is not giving us the optimal value.

Dynamic Programming Approach

- The optimal solution can be defined in terms of optimal sub-problems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say, the final multiplication is at index k: $(A_0 * \ldots * A_k) * (A_{k+1} * \ldots * A_{n-1})$.
- Let us consider all possible places for that final multiplication:
  - There are *n*-1 possible ***splits***. Assume we know the minimum cost of computing the matrix product of each combination $A_0 \ldots A_i$ and $A_{i+1} \ldots A_{n-1}$. Let's call these $N_{0,i}$ and $N_{i+1,n-1}$.
- Recall that A_i is a $d_i \times d_{i+1}$ dimensional matrix, and the final result will be a $d_0 \times d_n$.

Dynamic Programming Approach

- Define the following:

$$\boxed{N_{0,n-1} = \min_{0 \le k < n-1} \{N_{0,k} + N_{k+1,n-1} + d_0 d_{k+1} d_n\}}$$

- Then the optimal solution $N_{0,n-1}$ is the sum of two optimal sub-problems, $N_{0,k}$ and $N_{k+1,n-1}$ plus the time for the last multiplication.

Dynamic Programming Approach

香港科技大学(广州) THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Define **sub-problems**:
  - Find the best parenthesization of an arbitrary set of consecutive products: $A_i * A_{i+1} * \ldots * A_j$.
  - Let $N_{i,j}$ denote the **minimum** number of operations done by this sub-problem.
    - Define $N_{k,k} = 0$ for all k.
  - The optimal solution for the whole problem is then $N_{0,n-1}$.

Dynamic Programming Approach

- The characterizing equation for $N_{i,j}$ is:

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

- Note that, for example $N_{2,6}$ and $N_{3,7}$, both need solutions to $N_{3,6}$, $N_{4,6}$, $N_{5,6}$, and $N_{6,6}$. Solutions from the set of no matrix multiplies to four matrix multiplies.
  – This is an example of high sub-problem overlap, and clearly pre-computing these will significantly speed up the algorithm.

Recursive Approach

- We could implement the calculation of these $N_{i,j}$'s using a straightforward recursive implementation of the equation (aka not pre-compute them).

Algorithm ***RecursiveMatrixChain***(S, i, j):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

if i=j

  then return 0

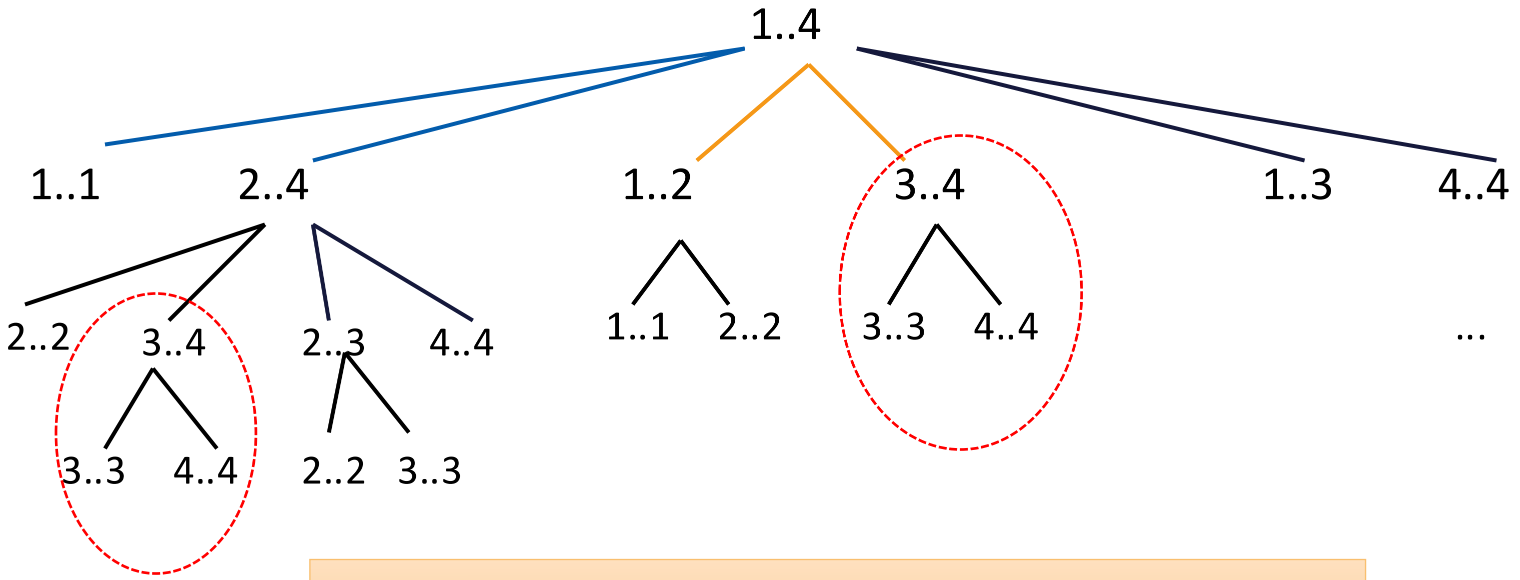
for $k \leftarrow i$ **to** j **do**

  $N_{i,j} \leftarrow \min\{N_{i,j},\ \textit{RecursiveMatrixChain}(S, i, k) + \textit{RecursiveMatrixChain}(S, k+1, j) + d_i d_{k+1} d_{j+1}\}$

return $N_{i,j}$

Subproblem Overlap

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + \ldots\}$$



How to schedule the sub-problems?

Dynamic Programming Algorithm

- High sub-problem overlap, with independent sub-problems indicate that a dynamic programming approach may work.
- Construct optimal sub-problems “bottom-up.” and remember them.
- $N_{i,i}$’s are easy, so start with them
- Then do problems of *length* 2,3,… sub-problems, and so on.
- Running time: $O(n^3)$

Dynamic Programming Algorithm

Algorithm *matrixChain*(S):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

for $i \leftarrow 1$ **to** $n - 1$ **do**

  $N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ **to** $n - 1$ **do**

  { $b = j - i$ is the length of the problem }

  for $i \leftarrow 0$ **to** $n - b - 1$ **do**

    $j \leftarrow i + b$

    $N_{i,j} \leftarrow +\infty$

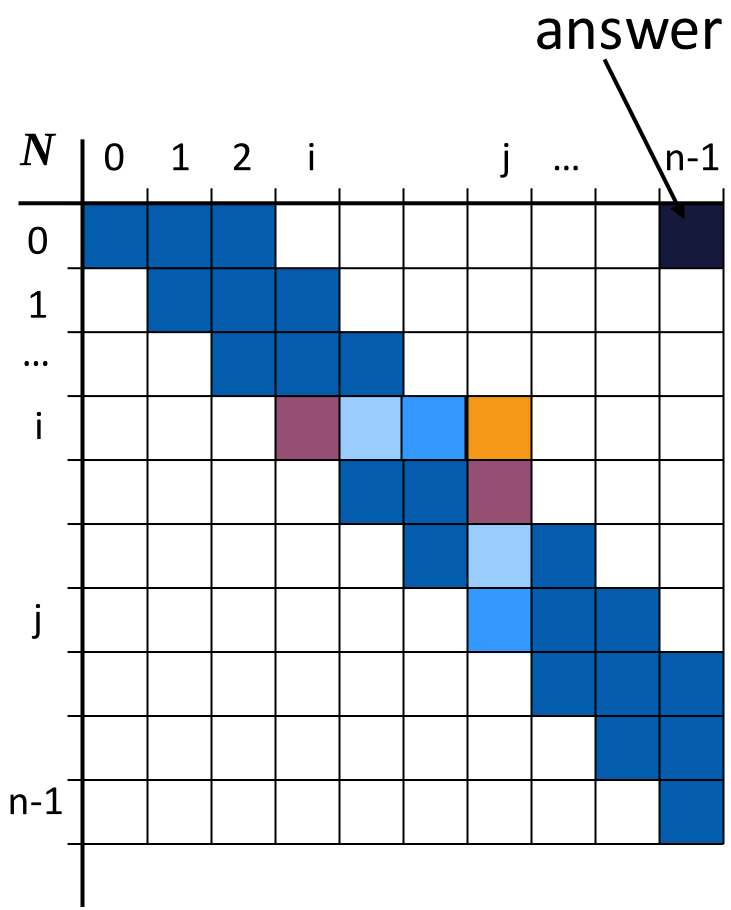
    for $k \leftarrow i$ **to** $j - 1$ **do**

      $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i\, d_{k+1}\, d_{j+1}\}$

return $N_{0,n-1}$

Algorithm Visualization

- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from previous entries in i-th row and j-th column
- Filling in each entry in the N table takes O(n) time.
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering "k" for each N entry



$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

Algorithm Visualization

- A_0: 30 X 35; A_1: 35 X15; A_2: 15X5;
  A_3: 5X10; A_4: 10X20; A_5: 20 X 25

0	1	2	3	4	5	
0	15,750	7,875	9,375	11,875	15,125	0
	0	2,625	4,375	7,125	10,500	1
		0	750	2,500	5,375	2
			0	1,000	3,500	3
				0	5,000	4
					0	5

$$N_{i,j} = \min_{i \le k < j}\{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

$$\begin{aligned}
N_{1,4} &= \min\{ \\
&N_{1,1} + N_{2,4} + d_1 d_2 d_5 = 0 + 2500 + 35*15*20 = 13000, \\
&N_{1,2} + N_{3,4} + d_1 d_3 d_5 = 2625 + 1000 + 35*5*20 = 7125, \\
&N_{1,3} + N_{4,4} + d_1 d_4 d_5 = 4375 + 0 + 35*10*20 = 11375 \\
&\} \\
&= 7125
\end{aligned}$$

Algorithm Visualization

$$(A_0*(A_1*A_2))*((A_3*A_4)*A_5)$$

	0	1	2	3	4	5
0	0	15,750	7,875	9,375	11,875	15,125
1		0	2,625	4,375	7,125	10,500
2			0	750	2,500	5,375
3				0	1,000	3,500
4					0	5,000
5						0

	0	1	2	3	4	5
0		0	0	2	2	2
1			1	2	2	2
2				2	2	2
3					3	4
4						4
5						

Matrix Chain-Products

- Some final thoughts
  - We ~~reduced~~ replaced a $\boldsymbol{O}(2^n)$ algorithm with a $\Theta(n^3)$ algorithm.
  - While the generic top-down recursive algorithm would have solved $\boldsymbol{O}(2^n)$ sub-problems, there are $\Theta(n^2)$ sub-problems.
    - Implies a high overlap of sub-problems.
  - The sub-problems are independent:
    - Solution to $A_0A_1\ldots A_k$ is independent of the solution to $A_{k+1}\ldots A_n$.

Matrix Chain-Products Summary

- Determine the cost of each pair-wise multiplication, then the ***minimum*** cost of multiplying three consecutive matrices (2 possible choices), using the pre-computed costs for two matrices.
- Repeat until we compute the minimum cost of all *n* matrices using the costs of the minimum *n*-1 matrix product costs.
  - *n*-1 possible choices.

Two Features of DP

- Optimal substructure
  - –an optimal solution to the problem contains within it optimal solutions to subproblems.
- Overlapping substructure
  - –the same subproblems are solved multiple times.

The 0/1 Knapsack Problem

- Given: A set S of *n* items (one piece each), with each item *i* having
  - w_i - a positive weight
  - b_i - a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case, we let T denote the set of items we take
  - Objective: maximize

$$\sum_{i \in T} b_i$$

Linear Programming formulation

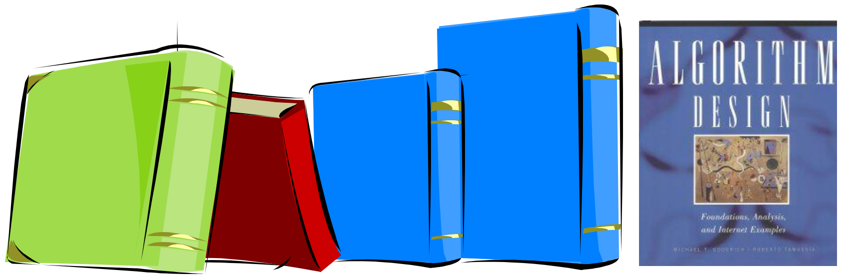
  - Constraint:

$$\sum_{i \in T} w_i \le W$$

Example

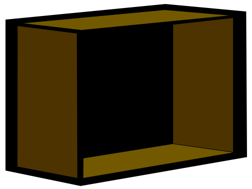
- Given: A set S of n items, with each item i having
  - b_i - a positive "benefit"
  - w_i - a positive "weight"
- Goal: Choose items with maximum total benefit but with weight at most W.

Items:



	1	2	3	4	5
Weight:	4 in	2 in	2 in	6 in	2 in
Benefit:	\$20	\$3	\$6	\$25	\$80

"knapsack"



box of width 9 in

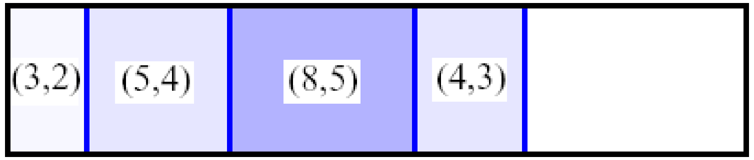
Solution:

- item 5 (\$80, 2 in)
- item 3 (\$6, 2 in)
- item 1 (\$20, 4 in)

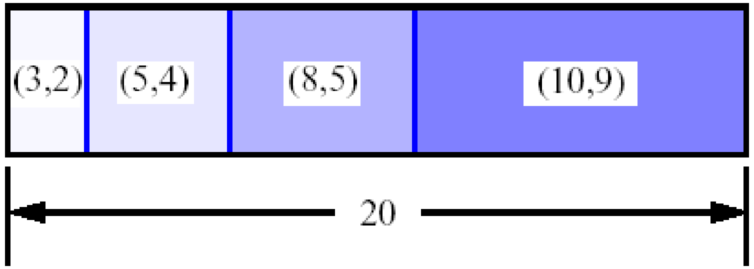
First Attempt

- S_k: Set of items numbered 1 to k.
- Define B[k] = best selection from S_k.
- Problem: does not have sub-problem optimality:
  - Consider set S={(3,2),(5,4),(8,5),(4,3),(10,9)} of (benefit, weight) pairs and total weight W = 20

Best for S_4:



Best for S_5:



Second Attempt

- S_k: Set of items numbered 1 to k.
- Define B[k,w] to be the best selection from S_k with weight at most *w*
- This does have sub-problem optimality.

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- I.e., the best subset of S_k with weight at most *w* is either:
  - the best subset of S_{k-1} with weight at most *w* or
  - the best subset of S_{k-1} with weight at most $w-w_k$ plus item *k*

Knapsack Example

Knapsack of capacity $W = 5$

$w_1 = 2, v_1 = 12 \quad w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20 \quad w_4 = 2, v_4 = 15$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

Max item allowed	Max Weight					
	0	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	**37**

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

Algorithm

- Since B[k,w] is defined in terms of B[k–1,*], we can use two arrays of instead of a matrix.
- Running time is **O**(nW).
- Not a polynomial-time algorithm since W may be large.
- Called a pseudo-polynomial time algorithm.

Algorithm

Input: set S of n items with benefit b_i and weight w_i; maximum weight W

Output: benefit of best subset of S with weight at most W

let A and B be arrays of length $W + 1$

for $w \leftarrow 0$ **to** W **do**
  $B[w] \leftarrow 0$
for $k \leftarrow 1$ **to** n **do**
  copy array B into array A
  for $w \leftarrow w_k$ **to** W **do**
    if $A[w-w_k] + b_k > A[w]$ **then**
      $B[w] \leftarrow A[w-w_k] + b_k$
return $B[W]$

All-pairs shortest paths

Input: Digraph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, with edge-weight function $w : E \to \mathrm{R}$.

Output: $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

IDEA:

- Run Bellman-Ford once from each vertex.

Bellman-Ford algorithm

Bellman-Ford*(G,s):

- $d^{(0)}[v] = \infty$ for all v in V
- $d^{(0)}[s] = 0$
- **For** i=0,…,n-1:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , \min_{u \text{ in } v.\text{inNeighbors}} \{d^{(i)}[u] + w(u,v)\})$
- If $d^{(n-1)} \neq d^{(n)}$:
  - **Return** NEGATIVE CYCLE ☹
- Otherwise, $\text{dist}(s,v) = d^{(n-1)}[v]$

Bellman-Ford is also an example of…

Dynamic Programming!

Running time: O(mn)

All-pairs shortest paths

Input: Digraph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, with edge-weight function $w : E \to \mathrm{R}$.

Output: $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

IDEA:

- Run Bellman-Ford once from each vertex.
- Time $= O(V^2E)$.
- Dense graph ($\Theta(n^2)$ edges) $\Rightarrow$ $\Theta(n^4)$ time in the worst case.

Good first try! Can we use DP to solve it?

Optimal substructure

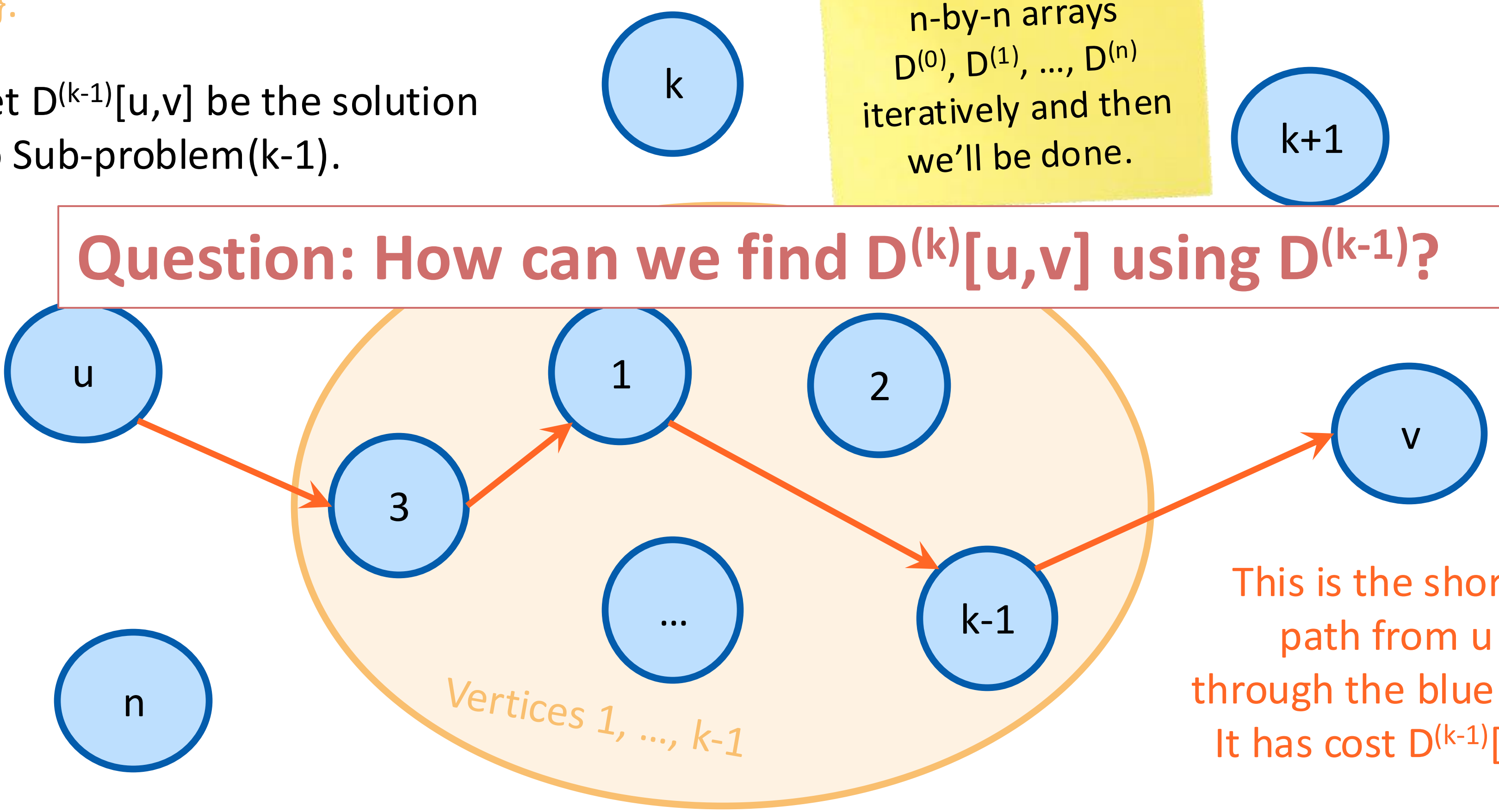
Sub-problem(k-1):
For all pairs, u,v, find the cost of the shortest path from u to v, so that all the internal vertices on that path are in {1,…,k-1}.

Let $D^{(k-1)}[u,v]$ be the solution to Sub-problem(k-1).

Our DP algorithm will fill in the n-by-n arrays $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$ iteratively and then we'll be done.

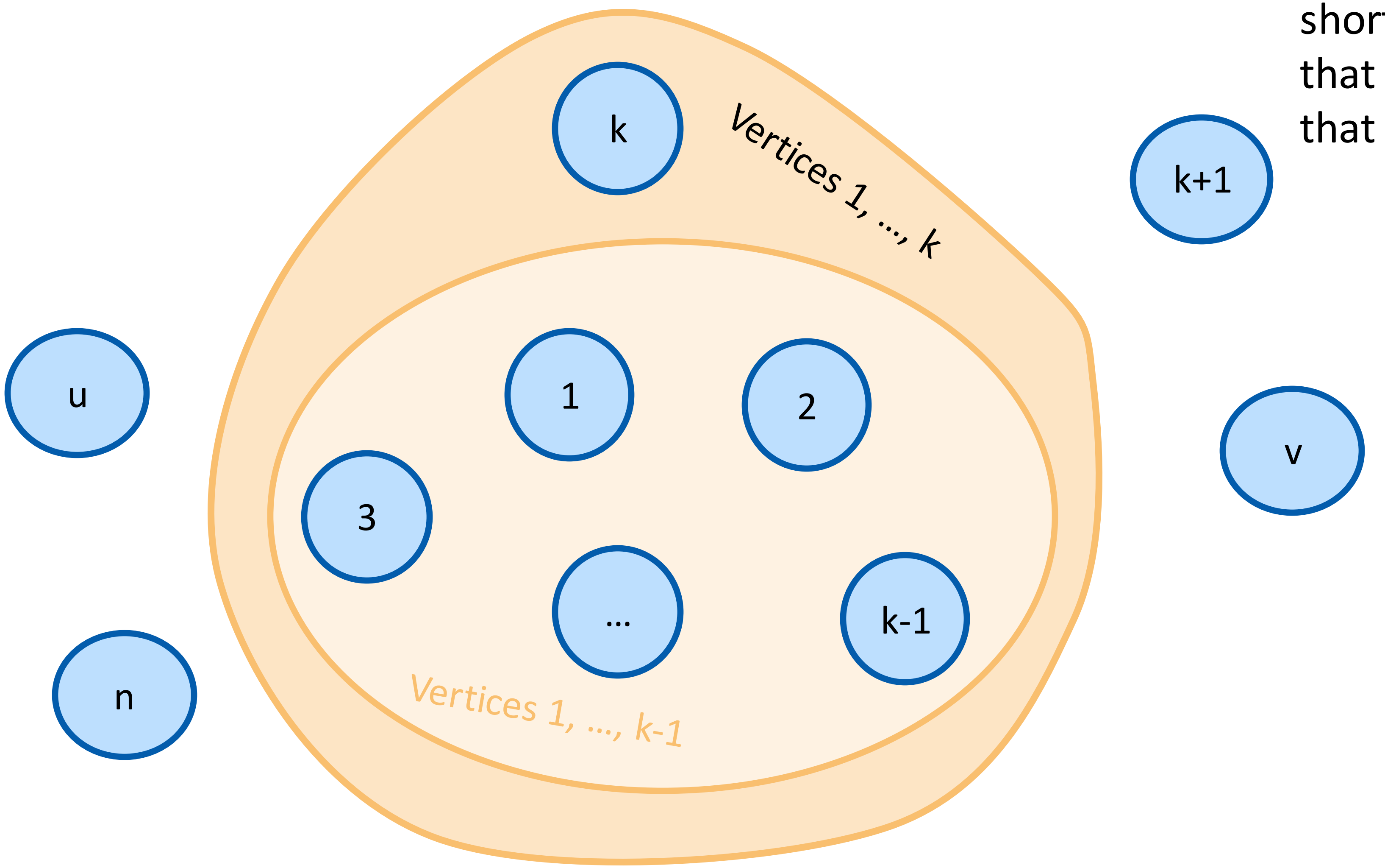
Label the vertices 1,2,…,n
(We omit some edges in the picture below – meant to be a cartoon, not an example).

Question: How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?



This is the shortest path from u to v through the blue set. It has cost $D^{(k-1)}[u,v]$

How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

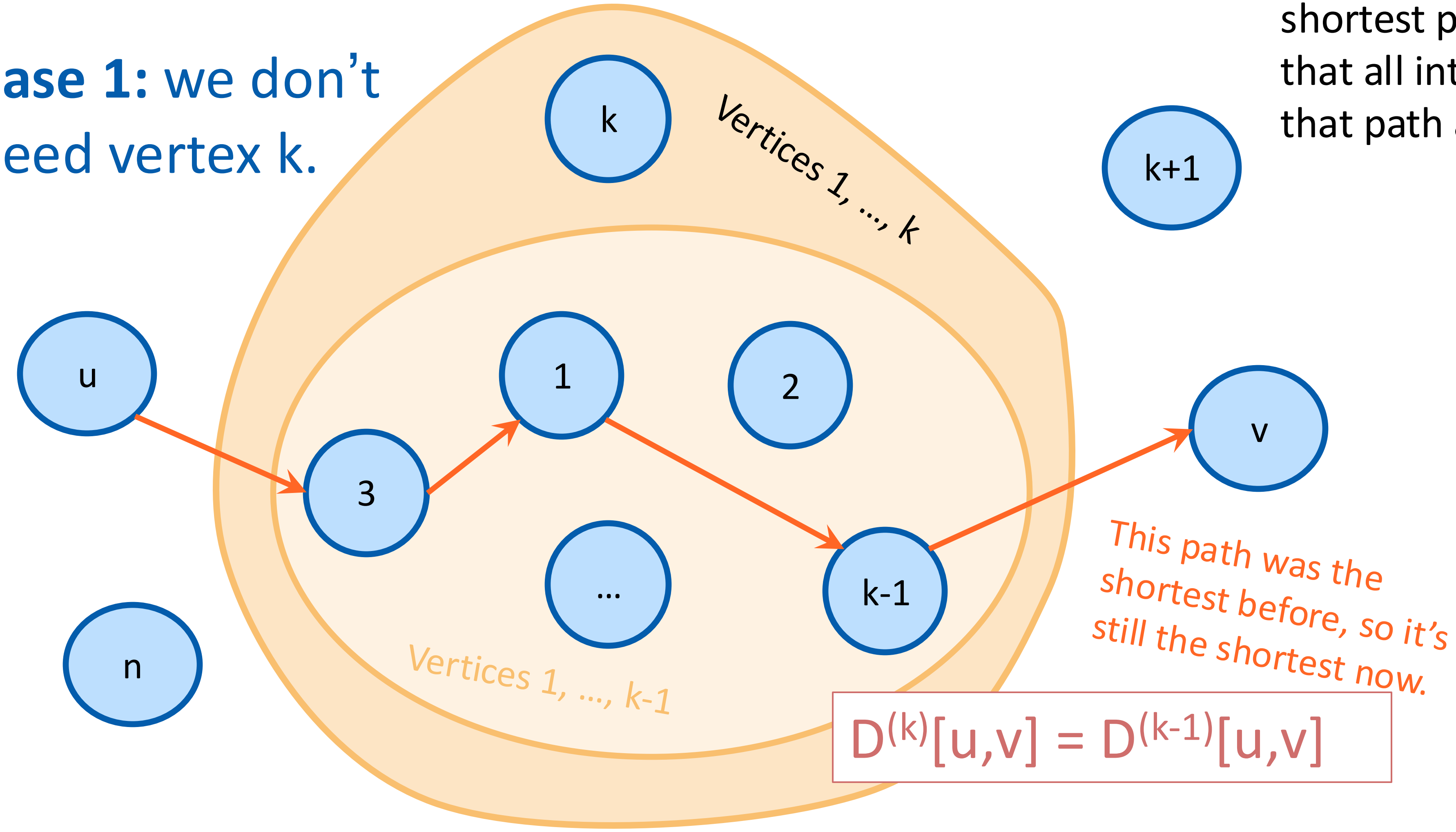


$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.

How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.

Case 1: we don't need vertex k.

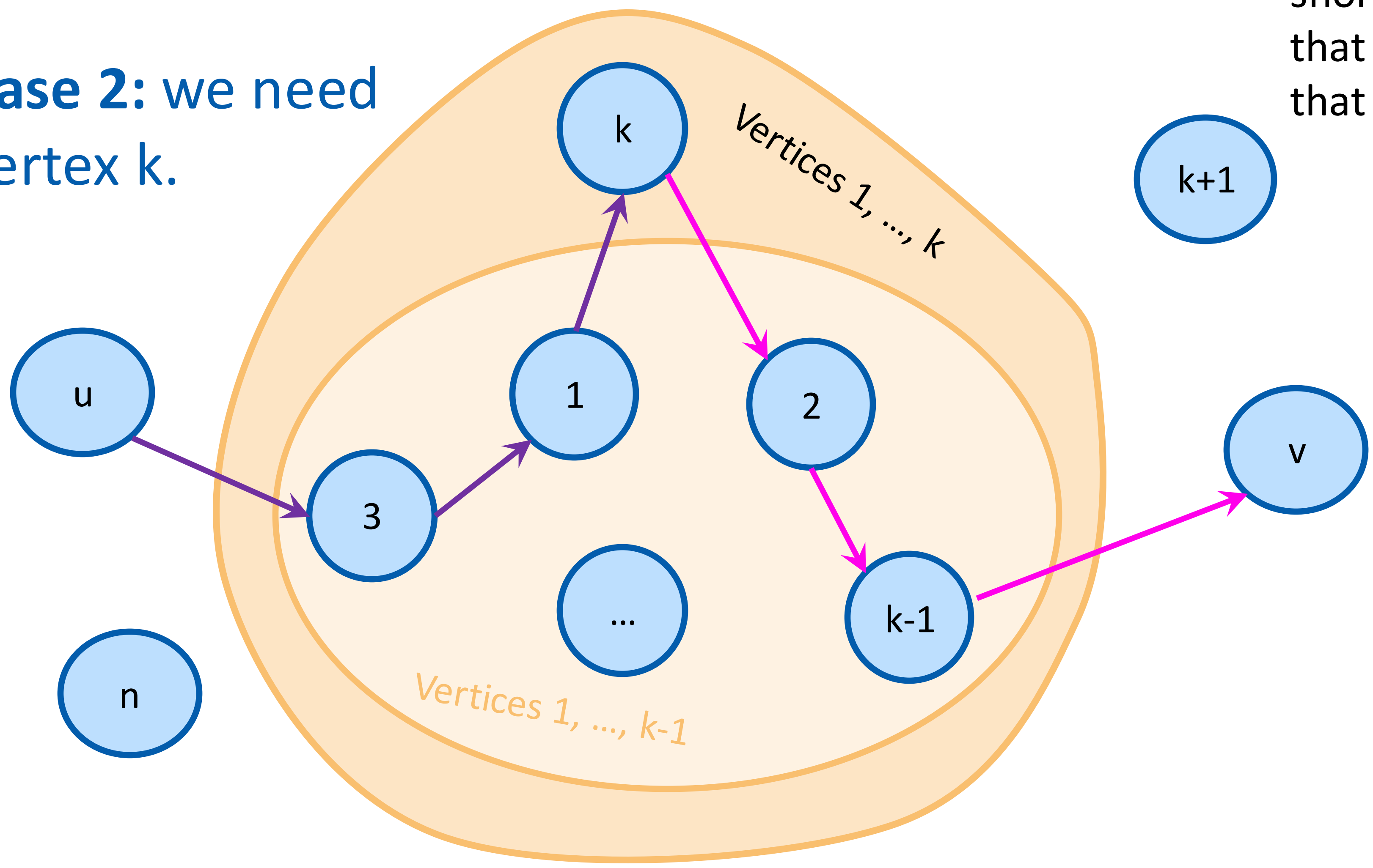


$$D^{(k)}[u,v] = D^{(k-1)}[u,v]$$

How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.

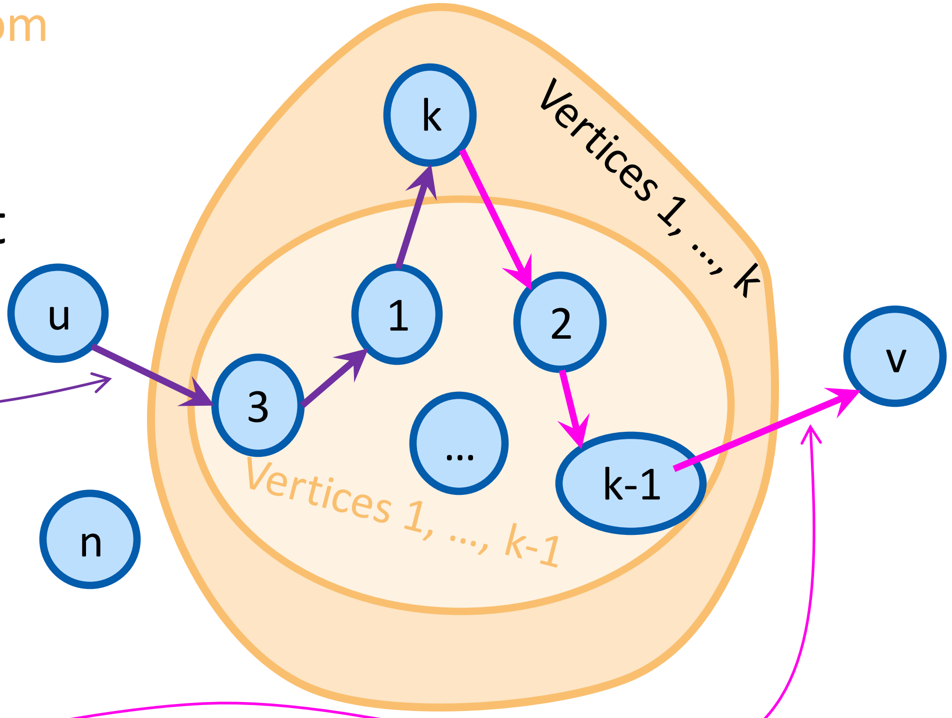
Case 2: we need vertex k.



Case 2 continued

- Suppose there are no negative cycles.
  - Then WLOG the shortest path from u to v through {1,…,k} is **simple**.
- If **that path** passes through k, it must look like this:
- **This path** is the shortest path from u to k through {1,…,k-1}.
  - sub-paths of shortest paths are shortest paths
- Similarly for **this path**.

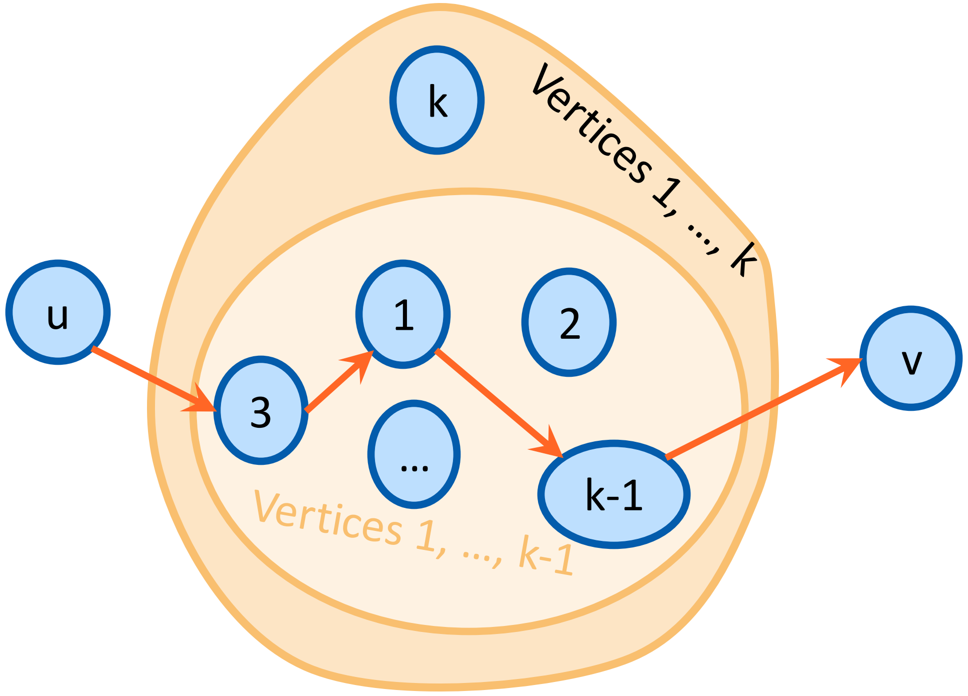
Case 2: we need vertex k.



$$D^{(k)}[u,v] = D^{(k-1)}[u,k] + D^{(k-1)}[k,v]$$

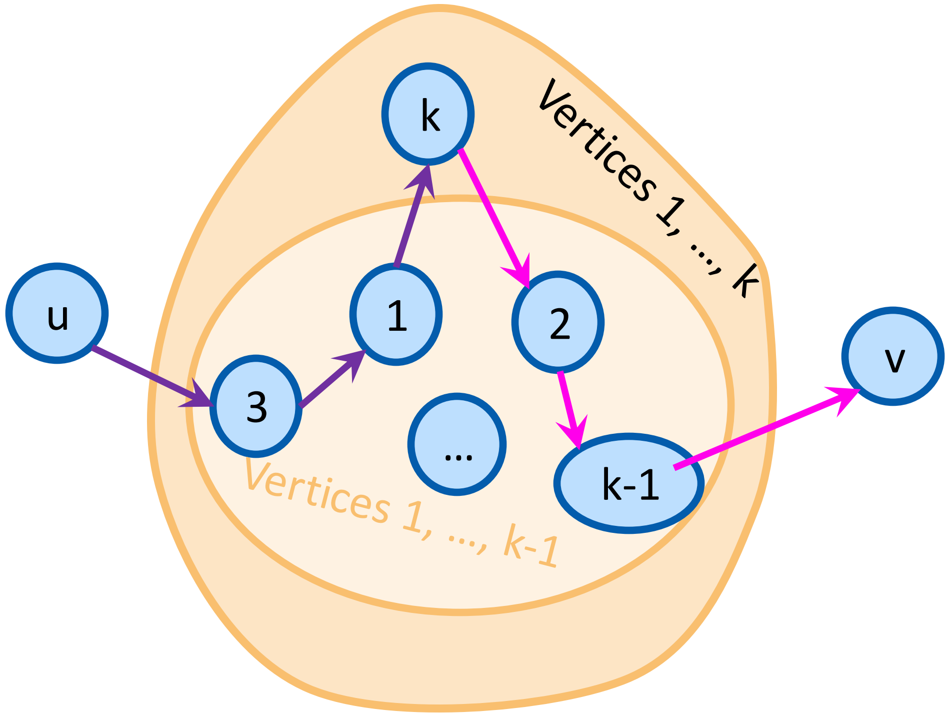
How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

Case 1: we don't need vertex k.



$$D^{(k)}[u,v] = D^{(k-1)}[u,v]$$

Case 2: we need vertex k.



$$D^{(k)}[u,v] = D^{(k-1)}[u,k] + D^{(k-1)}[k,v]$$

How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

- $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$

Case 1: Cost of shortest path through {1,…,k-1}

Case 2: Cost of shortest path from **u to k** and then from **k to v** through {1,…,k-1}

- Optimal substructure:
  – We can solve the big problem using solutions to smaller problems.
- Overlapping sub-problems:
  – $D^{(k-1)}[k,v]$ can be used to help compute $D^{(k)}[u,v]$ for lots of different u's.

How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

- $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$

Case 1: Cost of shortest path through {1,…,k-1}

Case 2: Cost of shortest path from **u to k** and then from **k to v** through {1,…,k-1}

- Using our ***Dynamic programming*** paradigm, this immediately gives us an algorithm!

Floyd-Warshall algorithm

- Initialize n-by-n arrays $D^{(k)}$ for k = 0,…,n
  - $D^{(k)}[u,u] = 0$ for all u, for all k
  - $D^{(k)}[u,v] = \infty$ for all $u \neq v$, for all k
  - $D^{(0)}[u,v] = \text{weight}(u,v)$ for all (u,v) in E.

The base case checks out: the only path through zero other vertices are edges directly from u to v.

- **For** k = 1, …, n:
  - **For** pairs u,v in V^2:
    - $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$
- **Return** $D^{(n)}$

This is a bottom-up *Dynamic programming* algorithm.

We’ve basically just shown

- Theorem:

  If there are no negative cycles in a weighted directed graph G, then the Floyd-Warshall algorithm, running on G, returns a matrix $D^{(n)}$ so that:

$$D^{(n)}[u,v] = \text{distance between u and v in G.}$$

- Running time: $O(n^3)$
  - Better than running Bellman-Ford n times!

Work out the details of a proof!

- Storage:
  - Need to store **two** n-by-n arrays, and the original graph.

As with Bellman-Ford, we don’t really need to store all n of the $D^{(k)}$.

What if there *are* negative cycles?

- Just like Bellman-Ford, Floyd-Warshall can detect negative cycles:
  - "Negative cycle" means that there's some v so that there is a path from v to v that has cost < 0.
  - Aka, $D^{(n)}[v,v] < 0$.

- Algorithm:
  - Run Floyd-Warshall as before.
  - If there is some v so that $D^{(n)}[v,v] < 0$:
    - **return** `negative cycle.`

What have we learned?

- The Floyd-Warshall algorithm is another example of *dynamic programming*.
- It computes All Pairs Shortest Paths in a directed weighted graph in time $O(n^3)$.

Can we do better than $O(n^3)$?

- There is an algorithm that runs in time $O(n^3/\log^{100}(n))$.
  - *[Williams, "Faster APSP via Circuit Complexity", STOC 2014]*
- If you can come up with an algorithm for All-Pairs-Shortest-Path that runs in time $O(n^{2.99})$, that would be a really big deal.
  - Let me know if you can!
  - See *[Abboud, Vassilevska-Williams, "Popular conjectures imply strong lower bounds for dynamic problems", FOCS 2014]* for some evidence that this is a very difficult problem!

Nothing on this slide is required knowledge in the exam!

Recap

- Two shortest-path algorithms:
  - Bellman-Ford for single-source shortest path
  - Floyd-Warshall for all-pairs shortest path
- ***Dynamic programming!***
  - This is a fancy name for:
    - Break up an optimization problem into smaller problems
      - The optimal solutions to the sub-problems should be sub-solutions to the original problem.
    - Build the optimal solution iteratively by filling in a table of sub-solutions.
      - Take advantage of overlapping sub-problems!

The End