Dynamic Programming

Chapter 10 of textbook 1

Outline

- Example of Dynamic programming:
  - Fibonacci numbers.
- What is dynamic programming, exactly?
  - And why is it called “dynamic programming”?
- Another example: Floyd-Warshall algorithm
  - An “all-pairs” shortest path algorithm

Fibonacci Numbers

- Definition:
  - F(n) = F(n-1) + F(n-2), with F(0) = F(1) = 1.
  - The first several are:

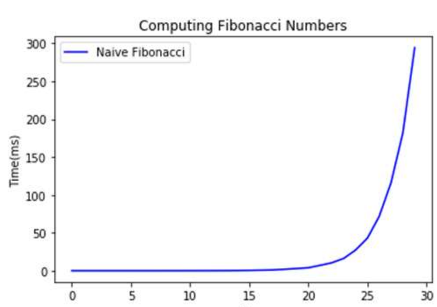
    1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,…
- Question:
  - Given n, what is F(n)?

Candidate algorithm

```
• def Fibonacci(n):
    • if n == 0 or n == 1:
        • return 1
    • return Fibonacci(n-1) + Fibonacci(n-2)
```

Running time?

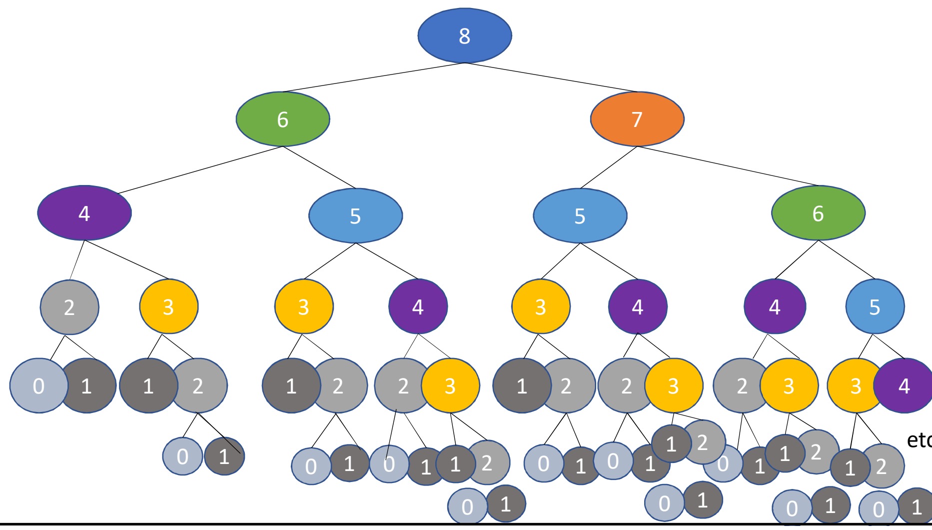
- T(n) = T(n-1) + T(n-2) + O(1)
- T(n) $\geq$ T(n-1) + T(n-2) for n $\geq$ 2
- So T(n) grows *at least* as fast as the Fibonacci numbers themselves…
- Fun fact, that's like α^n where $\alpha = \frac{1+\sqrt{5}}{2}$ is the golden ratio.
- aka, **EXPONENTIALLY QUICKLY** ☹



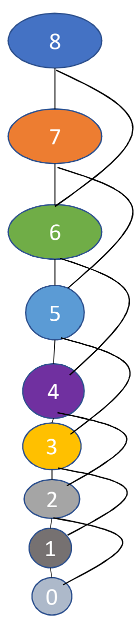
What's going on?

Consider Fib(8)

That's a lot of repeated computation!

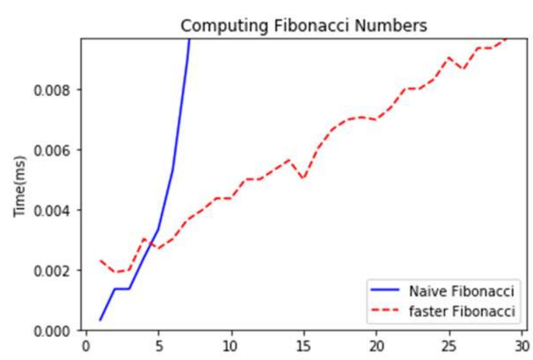


Maybe this would be better:



```
def fasterFibonacci(n):
    • F = [1, 1, None, None, …, None ]
        • \\ F has length n
    • for i = 2, …, n:
        • F[i] = F[i-1] + F[i-2]
    • return F[n]
```

Much better running time!



This was an example of…

Dynamic Programming!

What is *dynamic programming*?

- It is an algorithm design paradigm
  - like divide-and-conquer is an algorithm design paradigm.
- Usually it is for solving **optimization problems**
  - eg, **maximum** value, ***shortest*** path
  - (Fibonacci numbers aren't an optimization problem, but they are a good example…)

Elements of dynamic programming

1. Optimal sub-structure:

- Big problems break up into sub-problems.
  - Fibonacci: F(i) for $i \leq n$
- The solution to a problem can be expressed in terms of solutions to smaller sub-problems.
  - Fibonacci:

$$F(i+1) = F(i) + F(i-1)$$

Elements of dynamic programming

2. Overlapping sub-problems:

- The sub-problems overlap a lot.
  - Fibonacci:
    - Lots of different F[j] will use F[i].
  - This means that we can save time by solving a sub-problem just once and storing the answer.

Elements of dynamic programming

- Optimal substructure.
  - Optimal solutions to sub-problems are sub-solutions to the optimal solution of the original problem.
- Overlapping subproblems.
  - The subproblems show up again and again

- Using these properties, we can design a ***dynamic programming*** algorithm:
  - Keep a table of solutions to the smaller problems.
  - Use the solutions in the table to solve bigger problems.
  - At the end we can use information we collected along the way to find the solution to the whole thing.

Two ways to think about and/or implement DP algorithms

- Top down

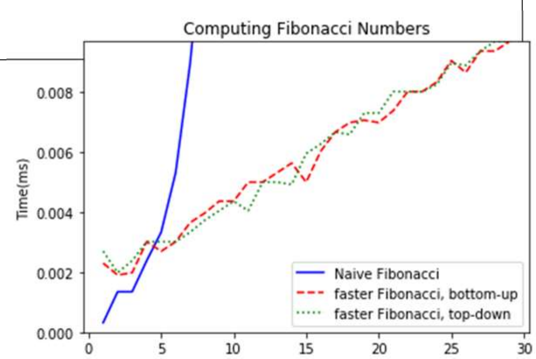
- Bottom up

Top down approach

- 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:
  - **Memo-ization**
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.

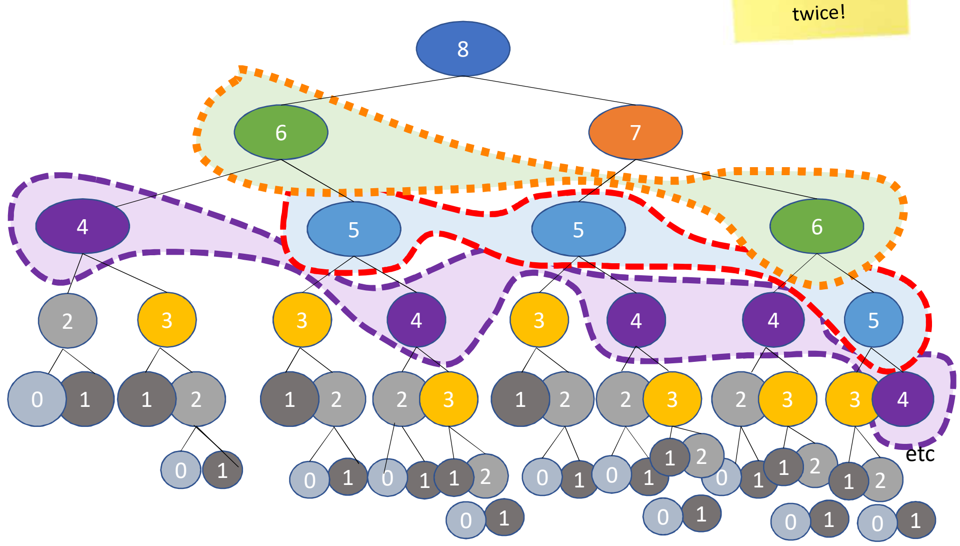
Example of top-down Fibonacci

```
• define a global list F = [1,1,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]
```



Memo-ization visualization

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

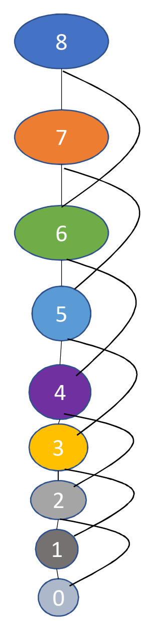


Memo-ization Visualization

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

But otherwise treat it like the same old recursive algorithm.

```
• define a global list F = [1,1,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]
```



Bottom up approach

- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
  - fill in F[2]
- …
- Then bigger problems
  - fill in F[n-1]
- Then finally solve the real problem.
  - fill in F[n]

Example of bottom-up approach

```
int Fibonacci( int N ){
        int i, Last, NextToLast, Answer;
        if( N <= 1 )
                return 1;

        Last = NextToLast = 1;
        for( i = 2; i <= N; i++){
                Answer = Last + NextToLast;
                NextToLast = Last;
                Last = Answer;
        }
    return Answer;
    }
```

Often the bottom up approach is simpler to write, and has less overhead, because you don't have to keep a recursive call stack

What have we learned?

- ***Dynamic programming:***
  - Paradigm in algorithm design.
  - Uses **optimal substructure**
  - Uses **overlapping subproblems**
  - Can be implemented **bottom-up** or **top-down**.
  - It’s a fancy name for a pretty common-sense idea:

Why “*dynamic programming*” ?

- Programming refers to finding the optimal “program.”
  - as in, a shortest route is a *plan* aka a *program*.
- Dynamic refers to the fact that it’s multi-stage.

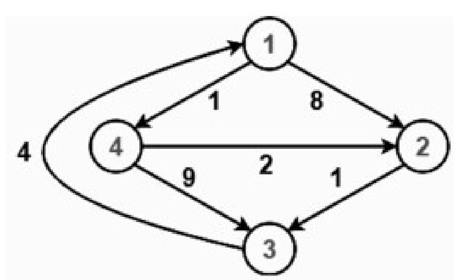
Why "*dynamic programming*" ?

- Richard Bellman invented the name in the 1950's.
- At the time, he was working for the RAND Corporation, which was basically working for the Air Force, and government projects needed flashy names to get funded.
- From Bellman's autobiography:
  - "It's impossible to use the word, dynamic, in the pejorative sense…I thought dynamic programming was a good name. It was something not even a Congressman could object to."

Floyd-Warshall Algorithm

Another example of DP

- This is an algorithm for **All-Pairs Shortest Paths** (APSP)
  - That is, I want to know the shortest path from u to v for **ALL pairs** u,v of vertices in the graph.
  - Not just from a special single source s.



Floyd-Warshall Algorithm
Another example of DP

- This is an algorithm for **All-Pairs Shortest Paths (**APSP)
  - That is, I want to know the shortest path from u to v for **ALL pairs** u,v of vertices in the graph.
  - Not just from a special single source s.
- Naïve solution:
  - For all s in G:
    - Run Dijkstra's algorithm on G starting at s.

Can we do in different ways?

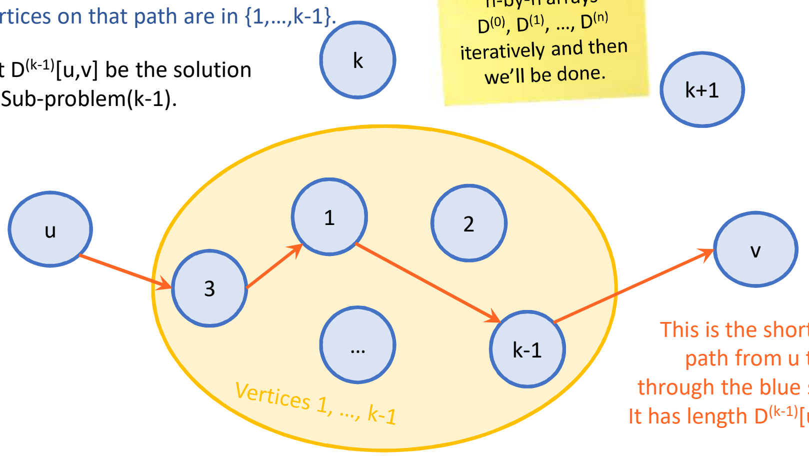
Optimal substructure

Label the vertices 1,2,…,n
(We omit some edges in the picture below).

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 $D^{(0)}$, $D^{(1)}$, …, $D^{(n)}$ iteratively and then we'll be done.



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

Optimal substructure

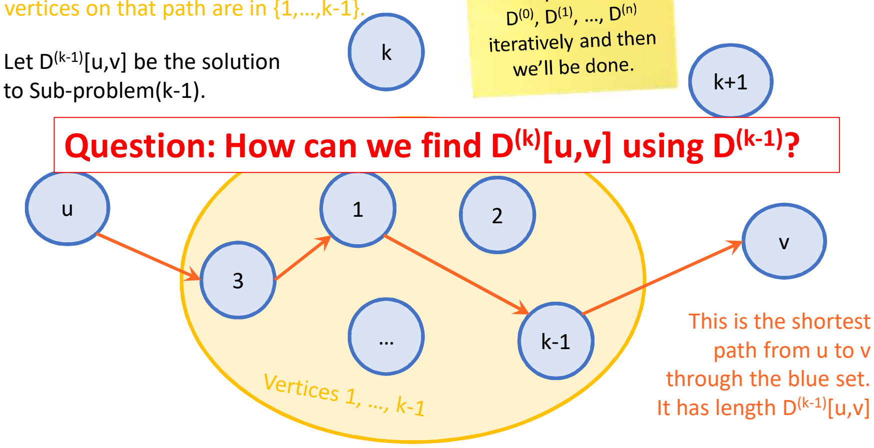
Label the vertices 1,2,…,n
(We omit some edges in the picture below).

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).

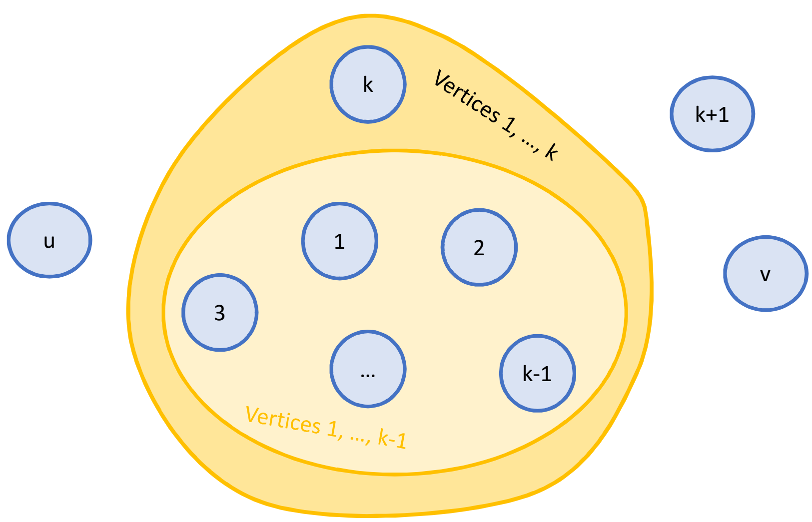
Our DP algorithm will fill in the n-by-n arrays $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$ iteratively and then we'll be done.

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



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

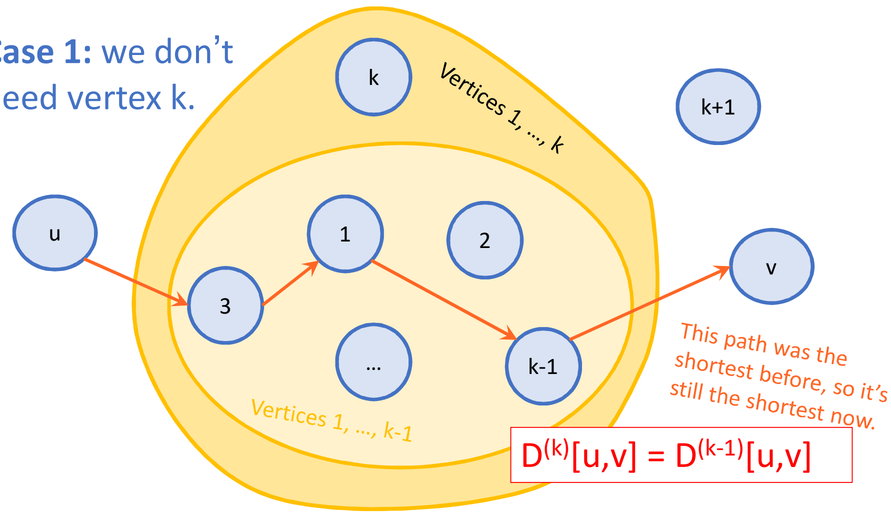
$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.



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}.

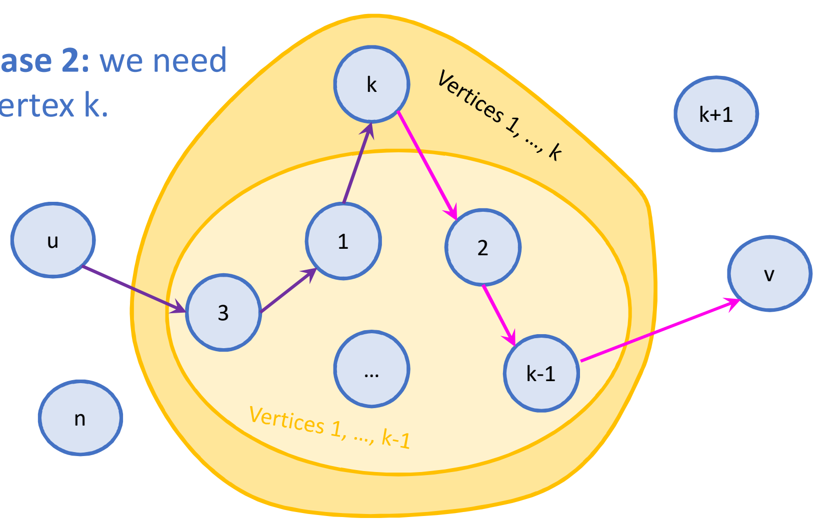
Case 1: we don't need vertex k.



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}.

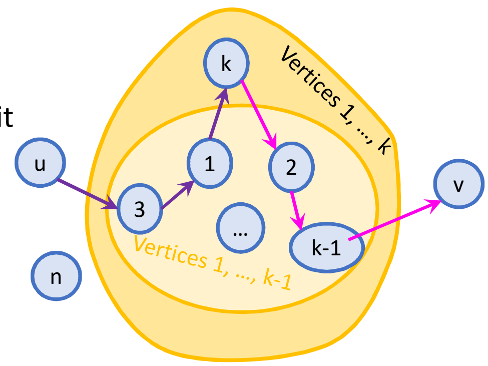
Case 2: we need vertex k.



Case 2 continued

- Suppose there are no negative edge
- 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

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 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

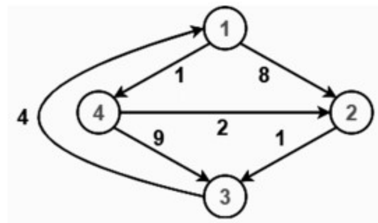
- Initialize n-by-n arrays $D^{(k)}$ for k = 0,…,n
  - $D^{(k)}[u,u] = 0$ for all u, for all k
  - $D^{(k)}[u,v] = \infty$ for all $u \neq v$, for all k
  - $D^{(0)}[u,v]$ = weight(u,v) for all (u,v) in E.
- **For** k = 1, …, n:
  - **For** pairs u,v in V^2:
    - $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$
- **Return** $D^{(n)}$

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

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

Example: initial

$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{array}$$



K = 1

- *Treat node **1** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*
  - *Distance[i][j] = minimum (Distance[i][j], Distance[i][**A**] + Distance[**A**][j])*
  - The elements in the first column and the first row are left as they are

$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

$$D_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \boxed{12} & 0 & \boxed{5} \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

K= 2

- *Treat node **2** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*
  - *Distance[i][j] = minimum (Distance[i][j], Distance[i][**A**] + Distance[**A**][j])*
  - The elements in the second column and the second row are left as they are

$$D_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \boxed{12} & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array} \qquad D_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \boxed{9} & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$

K=3

- Node 3

$$D_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \boxed{9} & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array} \qquad D_3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

K=4

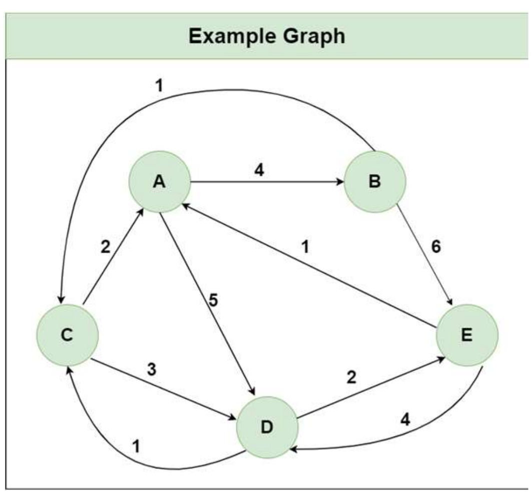
- Node 4

1+2=3

$3 < 8 \rightarrow 3$

$$D_3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array} \qquad D_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

Quiz



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:

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

- Running time: $O(n^3)$
  - Not really better than running Dijkstra n times.
    - But it's simpler to implement and handles negative weights.
- Storage:
  - Need to store **two** n-by-n arrays, and the original graph.

What if there *are* negative cycles?

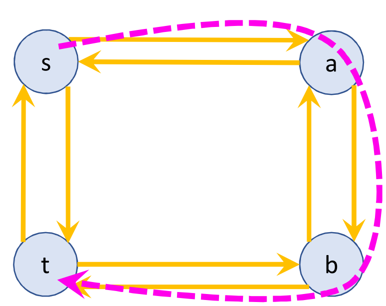
- Just like Bellman-Ford, Floyd-Warshall can detect negative cycles:
  - Negative cycle $\Leftrightarrow \exists$ v s.t. there is a path from v to v that goes through all n vertices that has cost < 0.
  - Negative cycle $\Leftrightarrow \exists$ v s.t. $D^{(n)}[v,v] < 0$.
- Algorithm:
  - Run Floyd-Warshall 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 $O(n^3)$.

Another Example of DP?

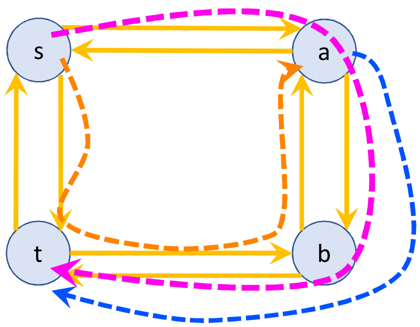
- Longest simple path (say all edge weights are 1):



What is the longest simple path from s to t?

This is an optimization problem…

- Can we use Dynamic Programming?
- Optimal Substructure?
  - Longest path from s to t = longest path from s to a
    + longest path from a to t?

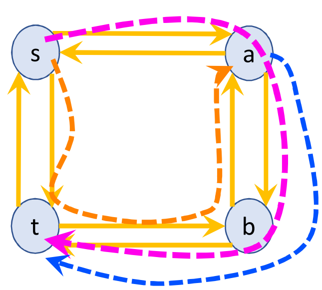


NOPE!

This doesn't give optimal sub-structure

Optimal solutions to subproblems don't give us an optimal solution to the big problem. (At least if we try to do it this way).

- The subproblems we came up with aren't independent:
  - Once we've chosen the longest path from a to t
    - which uses b,
  - our longest path from s to a shouldn't be allowed to use b
    - since b was already used.
- Actually, the longest simple path problem is NP-complete.
  - We don't know of any polynomial-time algorithms for it, DP or otherwise!



Recap

- ***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 (array) of sub-solutions.
      - Take advantage of overlapping sub-problems!

QUIZ (0-1 Knapsack problem)

Given a set of objects (items) which have both a value and a weight (vi, wi) what is the maximum value we can obtain by selecting a subset of these objects such that the sum of the weights does not exceed a certain capacity (knapsack capacity)

Note: The constraint here is we can either select an item completely or cannot select it at all [It is not possible to select a part of an item]. The number of every item in the original set is only 1.

Please apply dynamic programming to design an algorithm for this problem. Give your answer for this case:

Knapsack capacity = 7kg

	Object 1	Object 2	Object 3	Object 4	Object 5
Weight	1kg	2kg	3kg	3kg	4kg
Value	2$	3$	2$	4$	5$

Review of our course

- Algorithm:
  - Algorithm Analysis
  - Sorting Algorithm: Insertion Sort, Merge Sort, Quick Sort, Heap Sort
  - Algorithm Design Technique: Greedy Algorithm, Divide-and-Conquer Algorithm, Dynamic Programming
- Data structures:
  - Linked List: Single Linked List, Double Linked List,
  - Stack, Queue
  - Priority Queue (Heap)
  - Hashing
  - Tree
  - Graph

Final Exam

- Duration: 120 minutes
- Written exam (open-book): Printed materials are allowed. Electric devices are not allowed
- 5-6 questions:
  - Applying your knowledges to solve some problems.
  - You won't be required to write long C programs (we don't have enough time to do)
  - You will be asked to provide pseudo code or
  - You will modify or add some C code for existing programs