COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 27: Final Review II

Lecturer: Chris Lucas (cflucas@ncat.edu)

HW8!

Due @ 11:59PM ET

HW8!

Latest due date 12/04 @ 11:59PM ET

T-5 days until the Final!

12/06 from 2:00pm-4:00pm

T-5 days until the Final!

One double sided 8.5x11 cheat sheet!

Practice Final Solutions Released!

Big Questions!

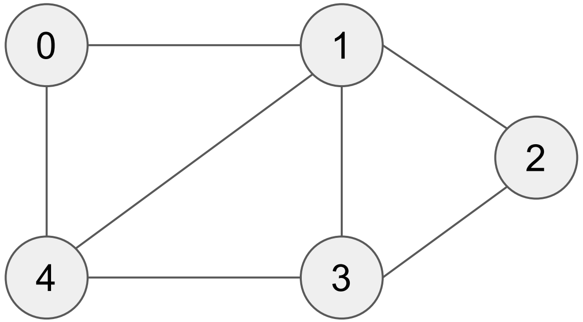
- Can we review what we've learned?
- What other questions do you have?

Big Questions!

- Can we review what we've learned?
- What other questions do you have?

Graphs

Graph Representation Examples



Graph

$$\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 \\
2 & 0 & 1 & 0 & 1 & 0 \\
3 & 0 & 1 & 1 & 0 & 1 \\
4 & 1 & 1 & 0 & 1 & 0
\end{array}$$

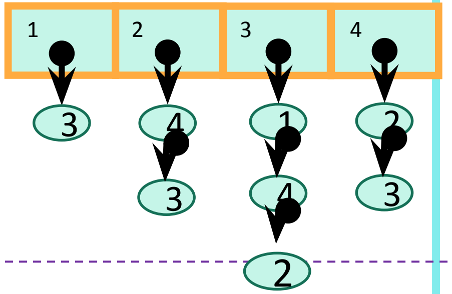
Adjacency Matrix

0: [1, 4]
1: [0, 4, 2, 3]
2: [1, 3]
3: [1, 4, 2]
4: [3, 0, 1]

Adjacency List

Generally better for **sparse** graphs (where $m \ll n^2$)

Trade-offs

Say there are V vertices and E edges.	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$	
Edge membership Is e = {v,w} in E?	O(1)	O(deg(V)) or O(deg(W))
Neighbor query Find all of v's neighbors.	O(V)	O(1)
Space requirements	O(V^2)	O(V + E)

Analogy

BFS

DFS

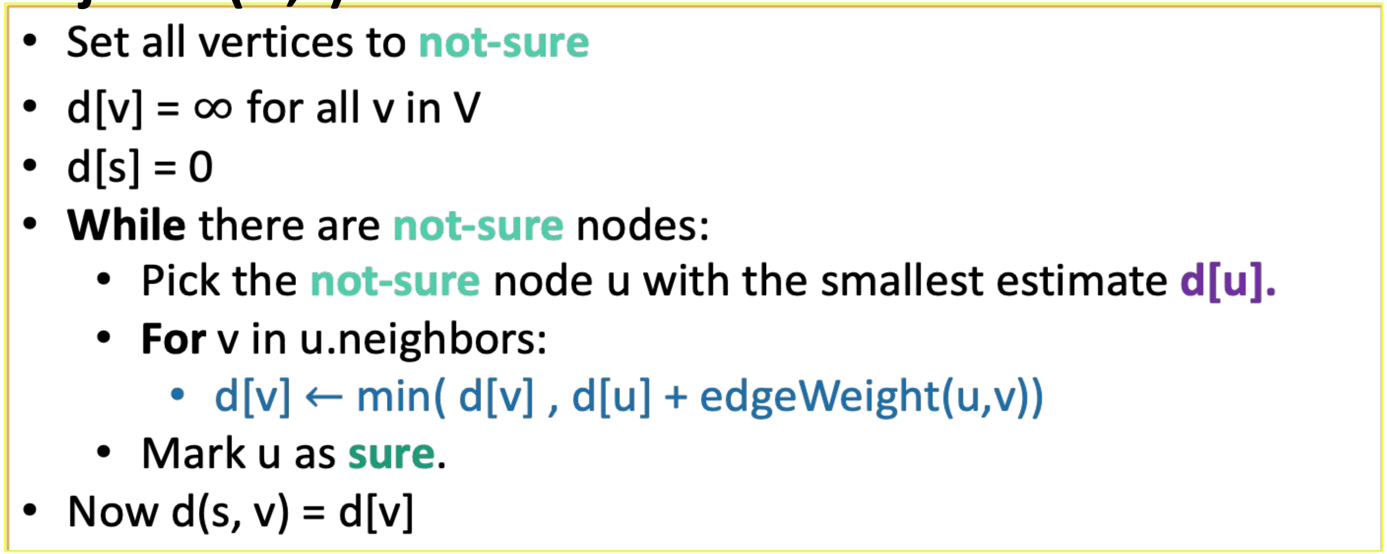
Breadth/Depth-First Search Pseudocode

```
algorithm BFS
   Input: undirected graph G = (V,E), s and d
   Output: true/false if path from s to d

    frontier = Queue of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

```
algorithm DFS
   Input: undirected graph G = (V,E), s and d
   Output: true/false if path from s to d

    frontier = Stack of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

https://visualgo.net/en/dfsbfs

Dijkstra's Running time?



Dijkstra(G,s):

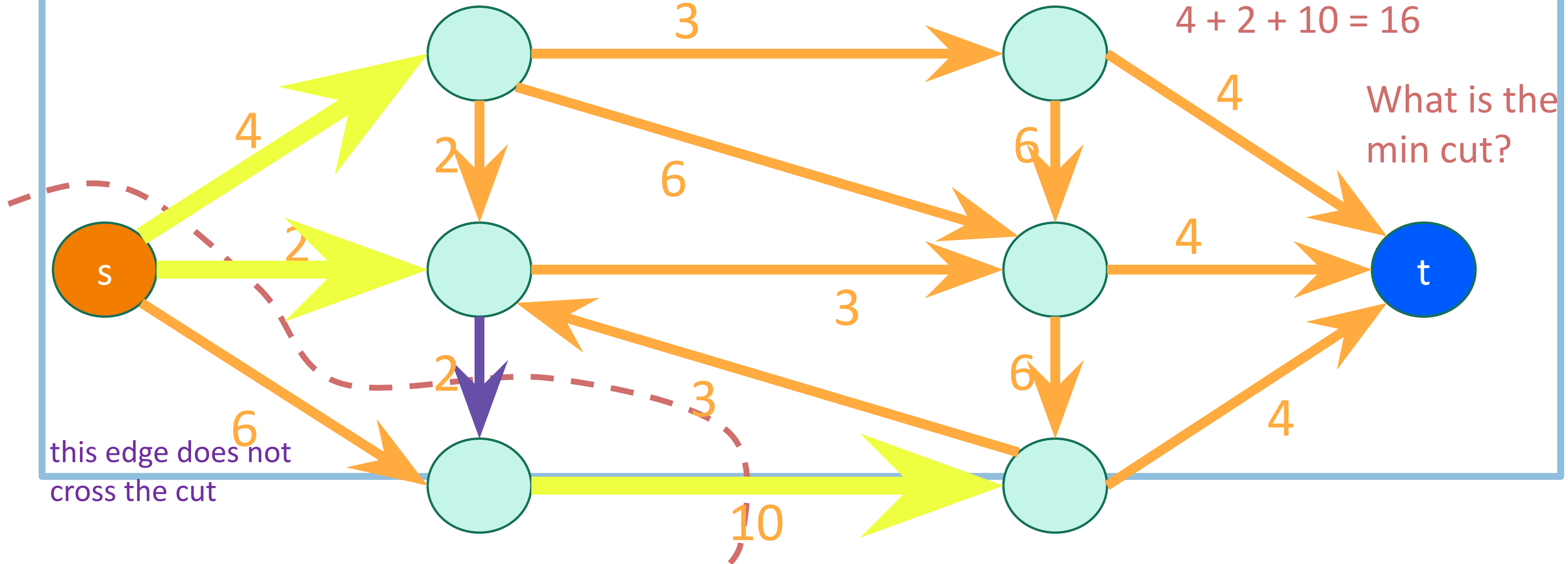
```
• Set all vertices to not-sure
• d[v] = ∞ for all v in V
• d[s] = 0
• While there are not-sure nodes:
    • Pick the not-sure node u with the smallest estimate d[u].
    • For v in u.neighbors:
        • d[v] ← min( d[v] , d[u] + edgeWeight(u,v))
    • Mark u as sure.
• Now d(s, v) = d[v]
```

- n iterations (one per vertex)
- How long does one iteration take?

Depends on how we implement it…

An s-t cut is a cut which separates s from t

- An edge **crosses the cut** if it goes from s's side to t's side.
- The **cost** (or capacity) of a cut is the sum of the capacities of the edges that cross the cut.



Flows

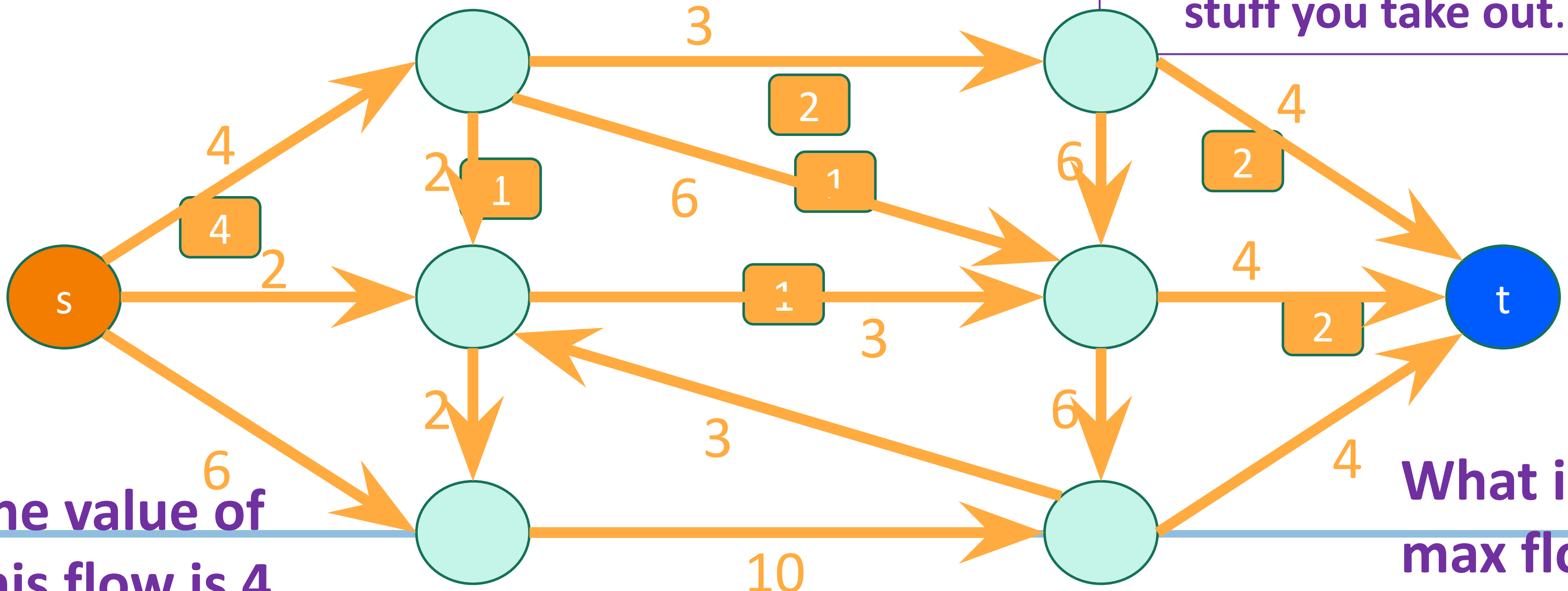
- The value of a flow is:
  - The amount of stuff coming out of s
  - The amount of stuff flowing into t

Because of conservation of flows at vertices,

stuff you put in

=

stuff you take out.



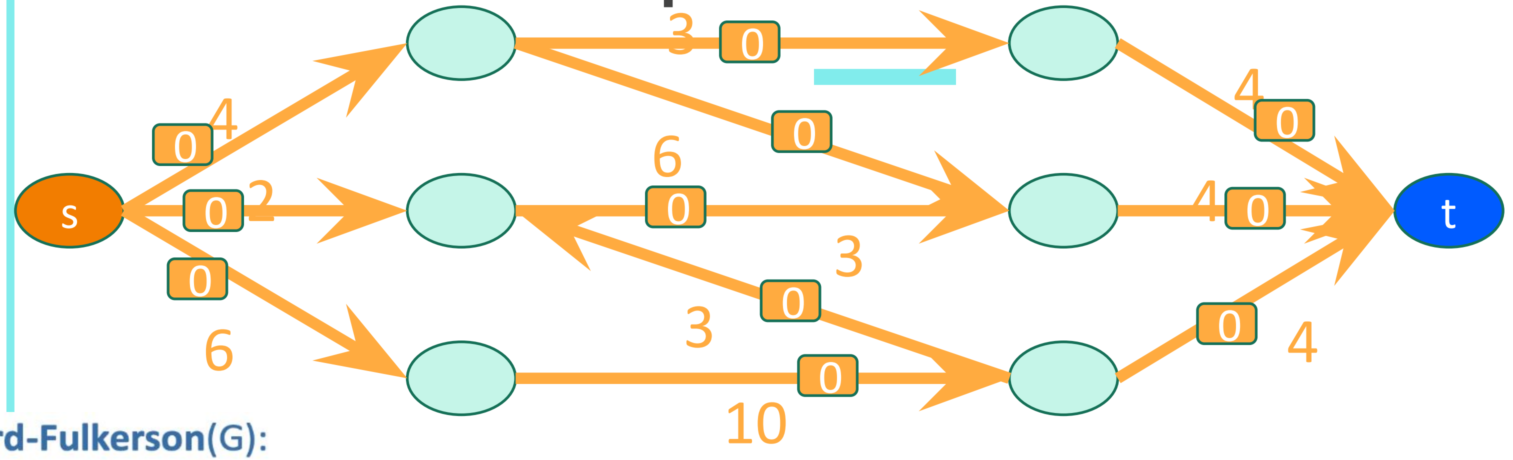
The value of this flow is 4.

What is the max flow?

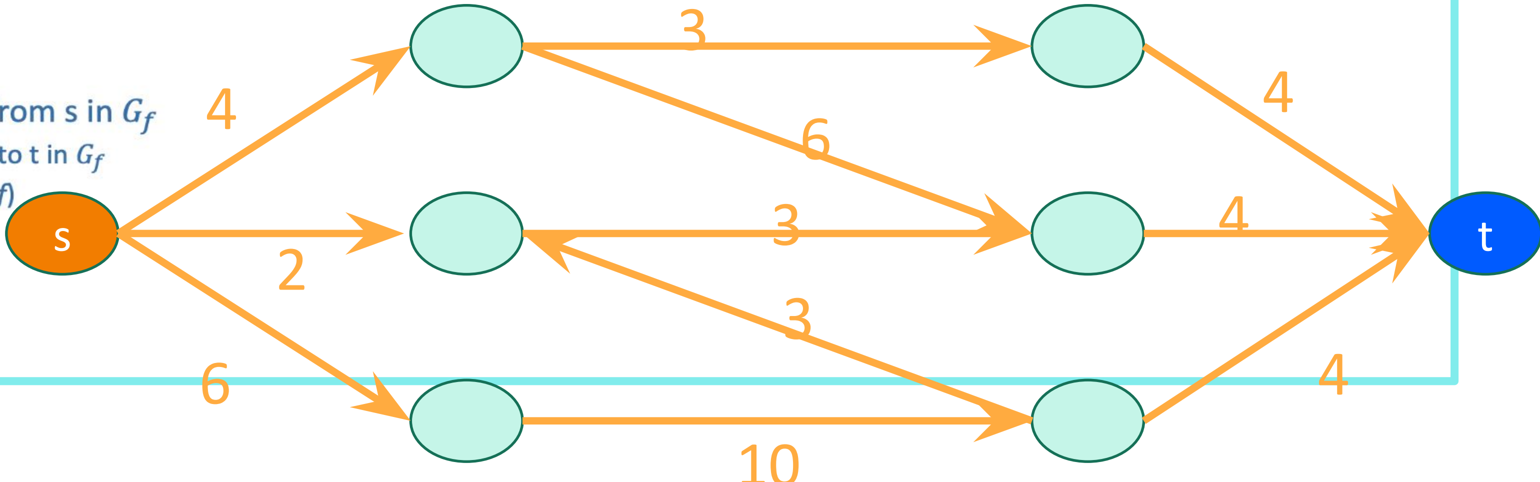
Ford-Fulkerson Algorithm

```
• Ford-Fulkerson(G):
    • $f \leftarrow$ all zero flow.
    • $G_f \leftarrow G$
    • while t is reachable from s in $G_f$
        • Find a path P from s to t in $G_f$        // e.g., use DFS or BFS
        • $f \leftarrow$ increaseFlow($P$,$f$)
        • update $G_f$
    • return $f$
```

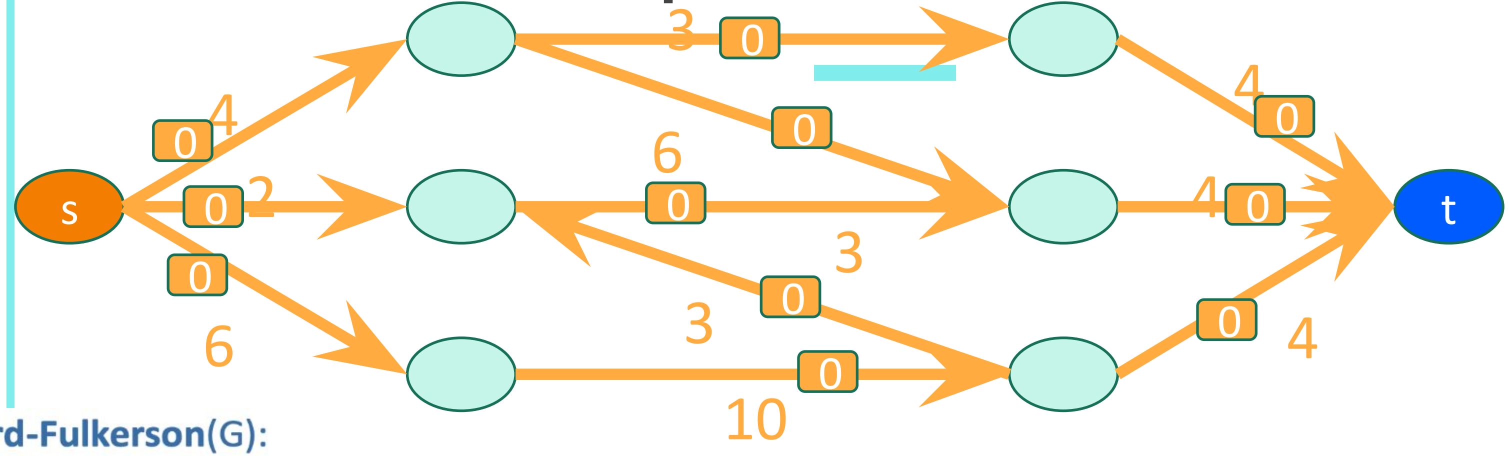
Example of Ford-Fulkerson



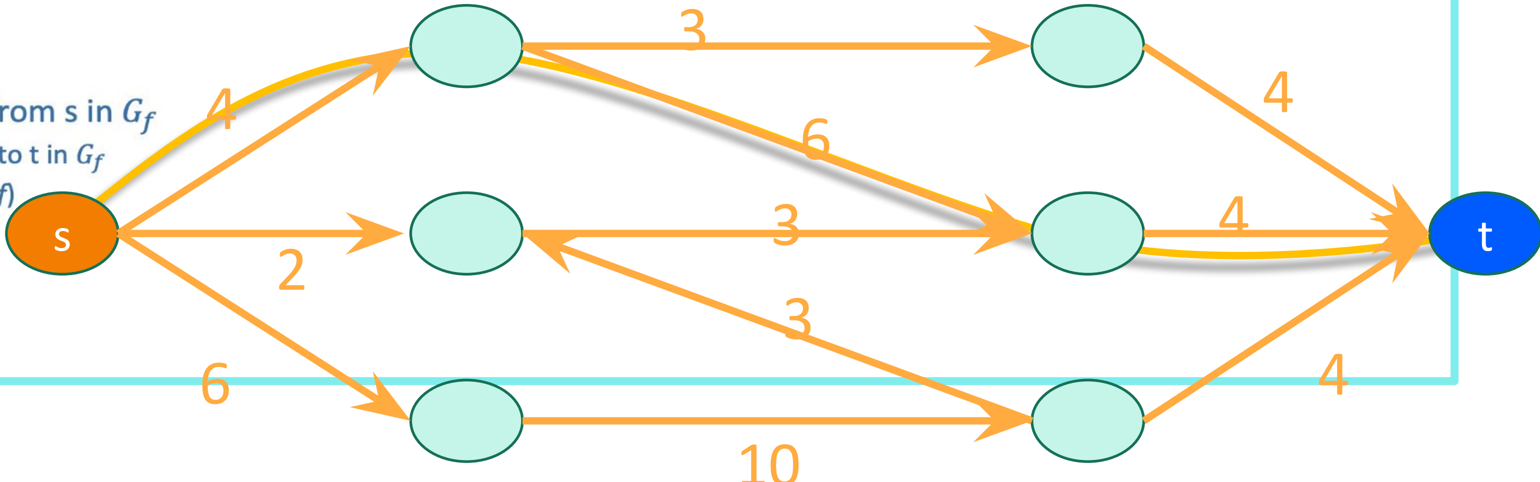
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



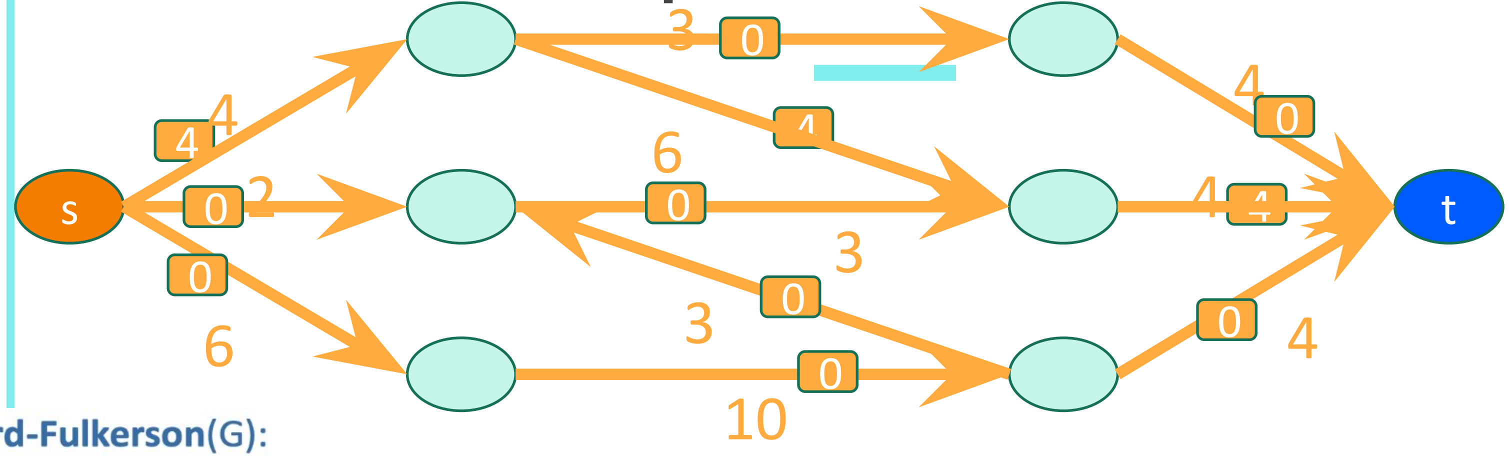
Example of Ford-Fulkerson

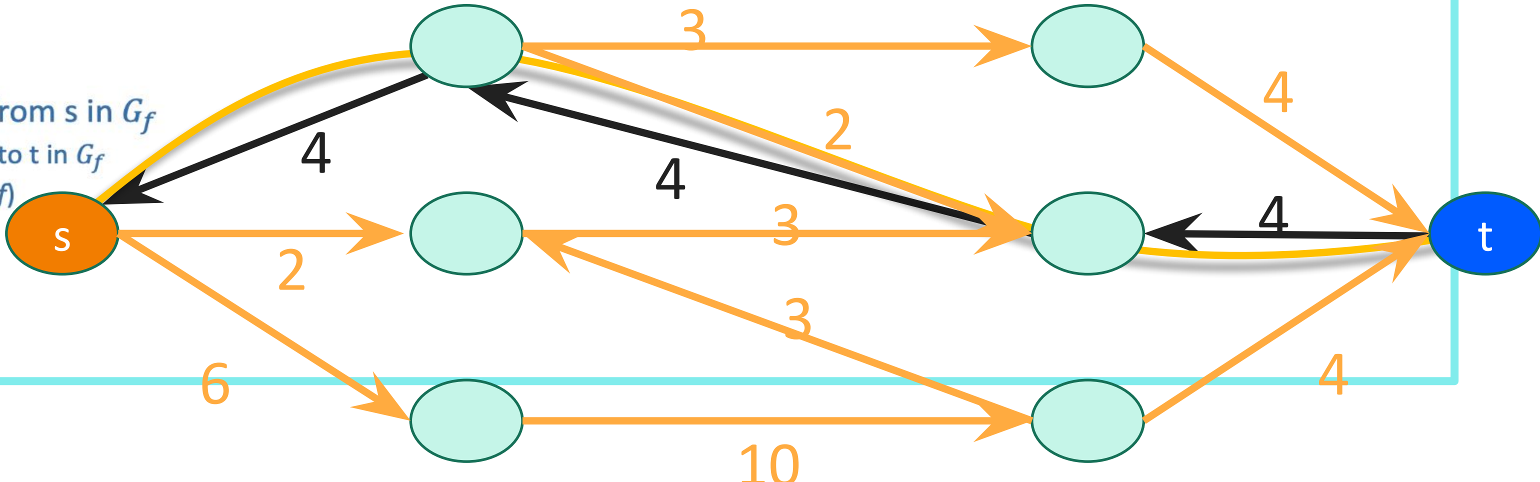


```
• Ford-Fulkerson(G):
    • $f \leftarrow$ all zero flow.
    • $G_f \leftarrow G$
    • while t is reachable from s in $G_f$
        • Find a path P from s to t in $G_f$
        • $f \leftarrow$ increaseFlow(P,f)
        • update $G_f$
    • return $f$
```



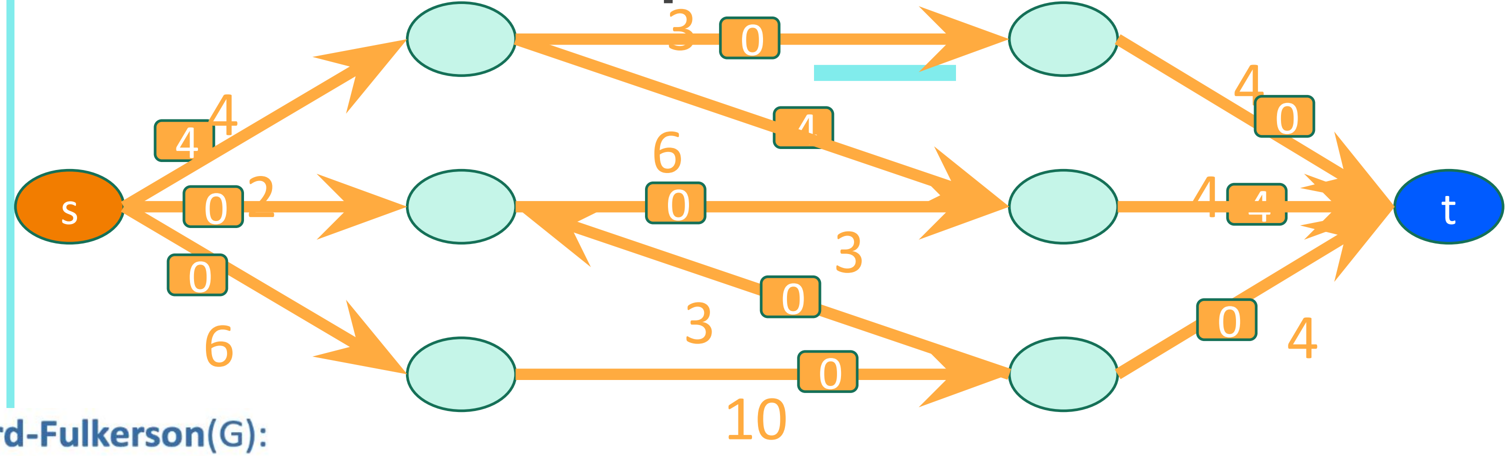
Example of Ford-Fulkerson



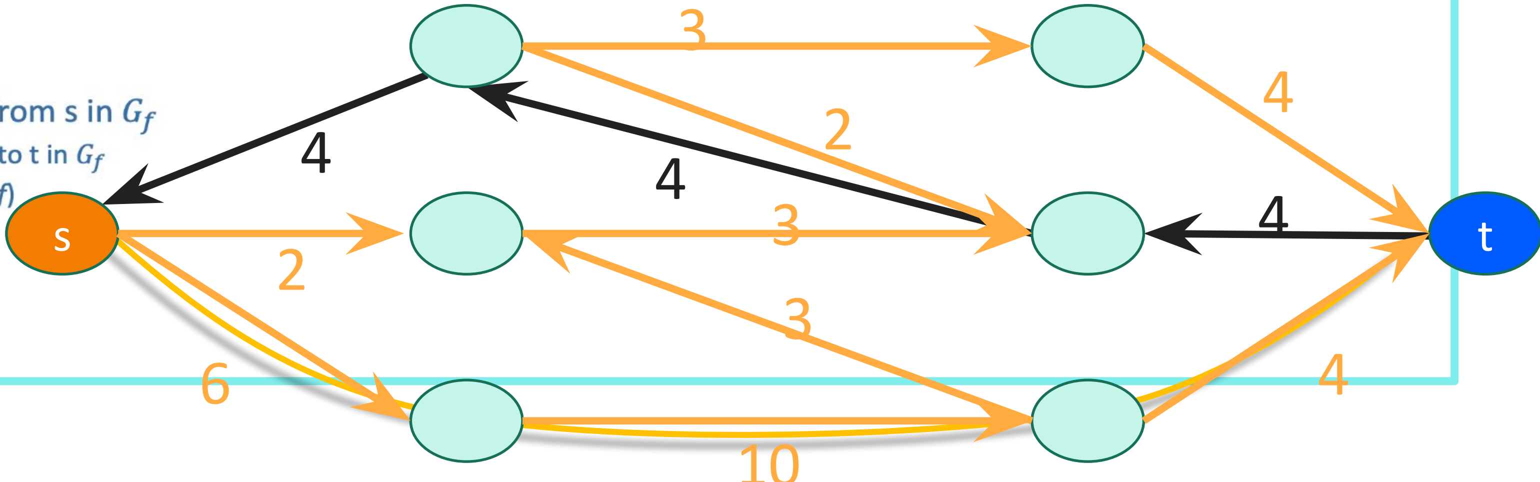


```
• Ford-Fulkerson(G):
    • $f \leftarrow$ all zero flow.
    • $G_f \leftarrow G$
    • while t is reachable from s in $G_f$
        • Find a path P from s to t in $G_f$
        • $f \leftarrow$ increaseFlow(P,f)
        • update $G_f$
    • return $f$
```

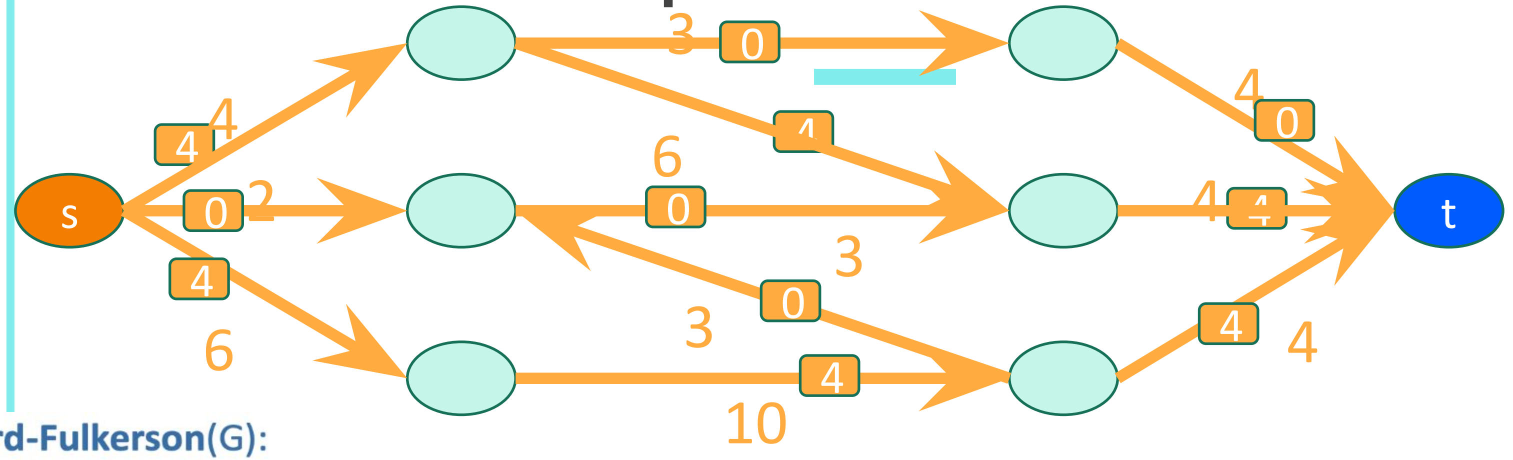
Example of Ford-Fulkerson



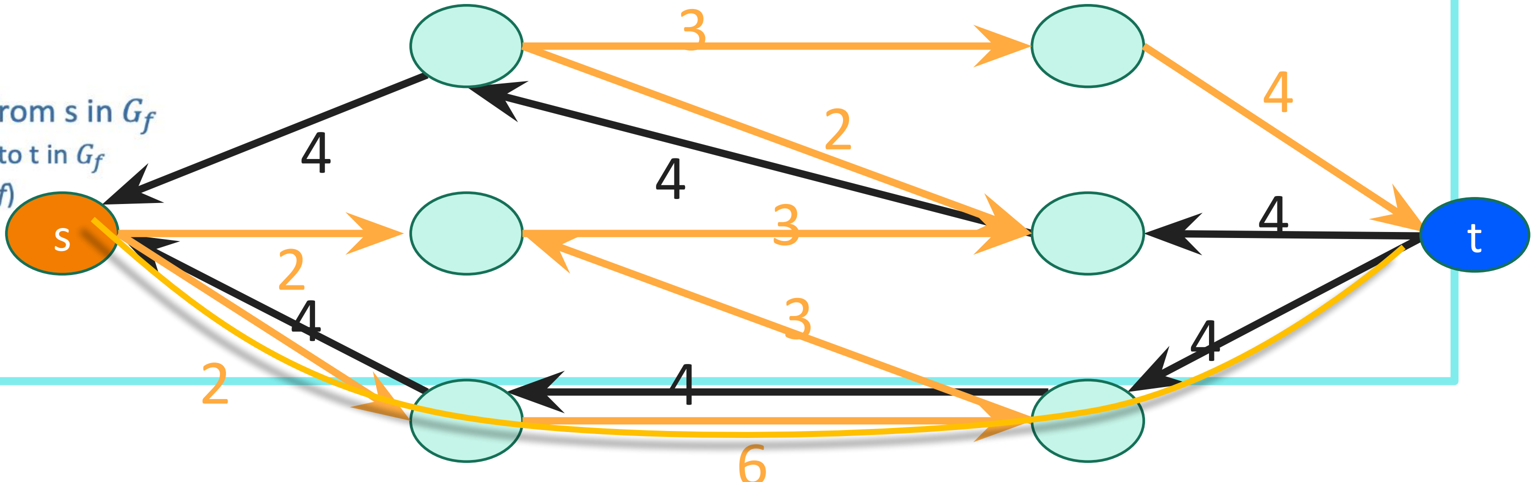
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



Example of Ford-Fulkerson



- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



Greediness/Dynamic Programming/ES&B!

Overview: What does “greedy” mean?

- Always makes the choice that seems to be the best at that moment (locally-optimal choice) in the hope that this will lead to a good solution overall (globally-optimal solution).
- In other words, we don't really plan, and we don't really look back. We just assume picking the next "obvious" best thing will lead to an optimal result. The key is often in knowing how to pick the next "obvious" best thing.
- This approach only works for some problems, so there is a skill in recognizing when a greedy approach will work.
- It often involves explicitly sorting, or using a data structure with sort-like behavior (e.g. priority queue, max/min heap)
- Examples that we’ve seen:
  - Dijkstra’s: pick next univisted node that’s closest based on dist.
  - Prim’s: pick next cheapest unvisited node to visit.
  - Kruskal’s: pick next cheapest edge that doesn’t cause a cycle.

Properties of Dynamic Programming

1. **Optimal substructure**
   - Big problems break up into sub-problems
     - **Fibonacci numbers: F(i) for i <= n**
   - The solution to a subproblem can be expressed in terms of solutions to smaller subproblems.
     - **Fibonacci numbers: F(i) = F(i-1) + F(i-2)**

2. **Overlapping subproblems**
   - Subproblems overlap/can be reused
     - **Fibonacci numbers:**
       1. **Both F[i+1] and F[i+2] directly use F[i]**
       2. **Lots of different F[i+x] indirectly use F[i].**
   - This means that we can save time by solving a sub-problem just once and storing the answer.
     - **To be continued…**

How to Create Algorithms with Dynamic Programming

1. Define recursive subproblem
   - What does an instance of the problem we're solving look like?

2. Relate subproblems
   - How do subproblems build upon or use other subproblems?

3. Top-down with memoization **or** build table bottom-up with ordering
   - e.g. Build table bottom-up by starting at i=1 then solving 2, 3, 4, … n

4. Solve original problem

Example: Longest Increasing Subsequence

```
algorithm longestIncreasingSubsequence
  Input: vector of integers vec of size N > 0
  Output: length of the longest increasing subsequence of vec

  L = array to store subproblem solutions
  for i = 0, 1, 2, 3, ... N-1:
    maxLength = 1
    for j = 0, 1, 2, ... i:
      if vec[j] < vec[i]
        maxLength = max(maxLength, L[j] + 1)
    L[i] = maxLength

  // find max
  answer = 1
  for each value in L:
     answer = max(value, answer)
  return answer
```

1. L[i] = longest subsequence ending at index i.
2. L[i] = max(L[j] for j in 0...i if vec[i] > vec[j]) + 1
3. Solve i = 0, 1, 2, … n
4. Return max value in table

Exhaustive Search & Backtracking

- Sometimes, the only way to solve a certain problem is through brute force, i.e. trying out every possible combination of values in order to get the correct answer. This process is called **exhaustive search**.
  - We have been dealing with $O(n^k)$ for the most part (polynomial time), but this approach gets into $O(2^n)$ which we've seen are very slow, but sometimes, it's the best we can do.
- We can reduce the cost in practice sometimes with **backtracking**, i.e. stopping early when we see we've hit a dead end while building our answer.
- The word "backtracking" is often colloquially used to refer to exhaustive search as well, even when there are no search constraints.

Exhaustive Search & Backtracking

- Write a program that prints out all n-character strings made up of A's, B's and C's
    - Input: integer n, representing the number of characters of the output strings
    - Output: print out all n-character strings using A, B, and C
    - Example: n = 2 => {aa, bb, cc, ab, ac, bc, ba, ca, cb}

1. **Choose**: What are we choosing at each step? What are we stepping over?
2. **Explore:** How will we modify the arguments before recursing?
3. **Unchoose:** How do we un-modify the arguments (if needed)?
4. **Base case:** What should we do when finished? How to know when finished?

Let’s code it!!!

Exhaustive Search & Backtracking

Write a program that prints out all n-character strings made up of A's, B's and C's

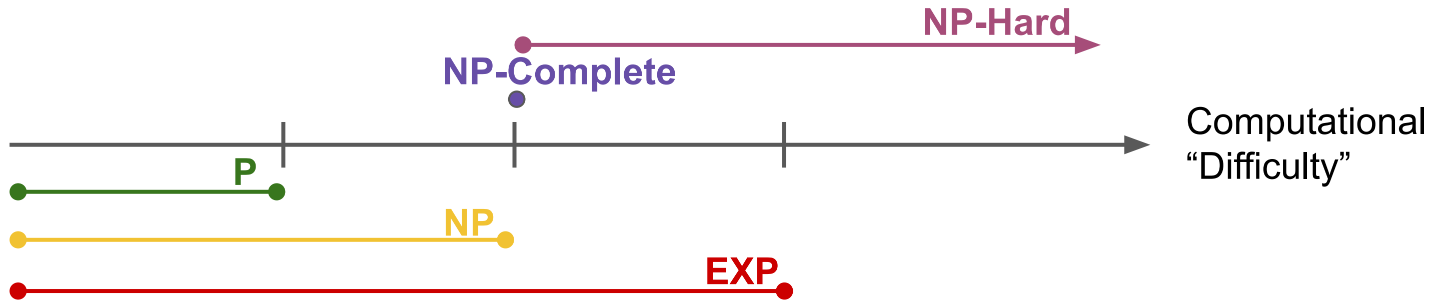
> Input: integer n, representing the number of characters of the output strings
>
> Output: print out all n-character strings using A, B, and C
>
> Example: n = 2 => {aa, bb, cc, ab, ac, bc, ba, ca, cb}

```
vector<string> alphabet = {"a", "b", "c"}
void nCharacterStrings(int n, string currString) {
        // Base case




        // Choose


        // Explore


        // Unchoose

}
```

Complexity Theory/Approximation Algorithms!

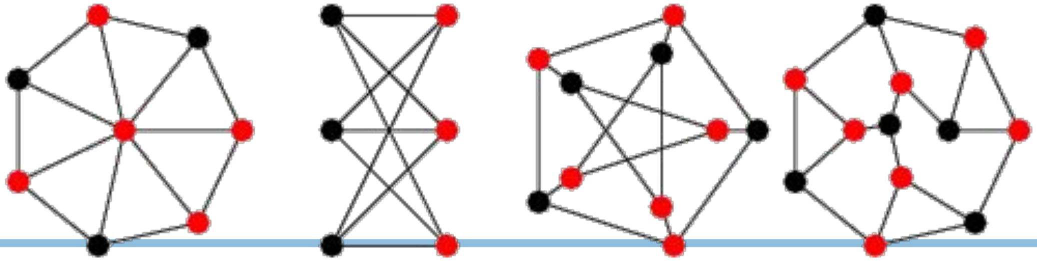
P, NP, EXP, NP-Hard, NP-Complete

- **P:** set of decision problems that can be solved in polynomial time
- **EXP**: set of decision problems that can be solved in exponential time
- **NP**: set of all decision problems that can be verified in polynomial time.
- **NP-Hard**: problems at least as hard as the hardest problems in NP.
- **NP-Complete**: problems that are NP-hard, but still in NP, i.e. "the hardest problems in NP".



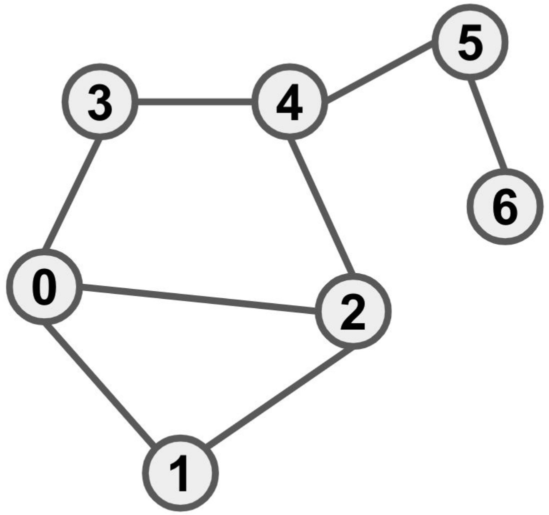
Vertex Cover

- Given a graph G = (V, E), a vertex cover is a set of nodes in G that touches every single edge in G at at least one end.
- A Minimum Vertex Cover of a graph is the smallest set of nodes possible to provide a vertex cover for a graph.
- Solving this optimally requires exponential time. The best we can do is try every possible vertex subset with exhaustive search. Can we approximate it?



Vertex Cover

```
algorithm generateApproximateVertexCover
  input: a graph G
  output: a vertex cover of G
initialize C to an empty set
while there are still edges in G
    pick uncovered edge (u, v) (smallest labeled)
    add u and v to vertex cover C
    delete all edges incident on u and v
return C
```



Takeaways

- $P \subseteq NP \subseteq EXP$
- NP-Complete problems are both NP-Hard and NP, and lots of interesting problems are NP-Complete. They can often be “reduced” to each other.
- P ?= NP asks whether the above two complexity classes are the same. It is likely not true, but has not been proven.
- The smallest change in a problem statement can make it P or NP, and it is not immediately obvious: MST versus Traveling Salesman versus Minimum Vertex Cover
- When we realize a problem will likely only have an exponential solution, we can come up with an algorithm that will give us an answer that is “good enough”

Big Questions!

- Can we review what we’ve learned?
- What other questions do you have?

Career Advice (High-level)

- Know what you want out of your career and where you're headed
    - But remain flexible should plans, thoughts, desires change…
- Get good at communicating your work upwards and outwards
    - No one will advocate for you and your career more intensely than you will
- Work ethic > raw intelligence
    - Focus on what's in your control
- Find a mentor
    - Preferably within your company, not someone you report to and someone you trust, can speak candidly to
- Healthy level of "selfishness"
    - (But don't totally suck…)

Career Advice (Slightly more tactical)

- Managing up is a real thing
  - Come with solutions not problems, but don't solve problems quietly
- Communicate your goals to your manager
  - Be explicit, set timelines, and follow up conversations against those goals. You can't improve what you don't measure
- Ask for feedback often
  - But *especially* after you do something well
- Document your achievements, feedback received, etc. as you go
  - Useful for performance reviews, later in life (grad school?)
- Be decisive and don't apologize
  - Change jobs, switch teams, do what you need to do for your career
- Schedule send lol

Thank you!

And best of luck on the final (and what comes thereafter)!

linkedin.com/in/chrisflucas

COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 27: Final Review II

Lecturer: Chris Lucas (cflucas@ncat.edu)