

BLG 336E

Analysis of Algorithms II

Lecture 13:
NP-Completeness and Review

Basic reduction strategies

1. Reduction by simple equivalence.
2. Reduction from special case to general case.
3. Reduction by encoding with gadgets.

1. **Claim.** VERTEX-COVER \equiv_p INDEPENDENT-SET.

Pf. We show S is an independent set iff $V - S$ is a vertex cover.

2. **Claim.** VERTEX-COVER \leq_p SET-COVER.

Pf. Given a VERTEX-COVER instance $G = (V, E)$, k , we construct a set cover instance whose size equals the size of the vertex cover instance.

3. **Claim.** 3-SAT \leq_p INDEPENDENT-SET.

Pf. Given an instance Φ of 3-SAT, we construct an instance (G, k) of INDEPENDENT-SET that has an independent set of size k iff Φ is satisfiable.

Polynomial-Time Reductions



constraint satisfaction

3-SAT reduces to
INDEPENDENT SET

3-SAT

Dick Karp
(1972)
1985 Turing
Award

INDEPENDENT SET

DIR-HAM-CYCLE

GRAPH 3-COLOR

SUBSET-SUM

VERTEX COVER

HAM-CYCLE

PLANAR 3-COLOR

SCHEDULING

SET COVER

TSP

packing and covering

sequencing

partitioning

numerical

Decision Problems

Decision problem.

- X is a set of strings.
- Instance: string s .
- Algorithm A solves problem X : $A(s) = \text{yes}$ iff $s \in X$.

Polynomial time. Algorithm A runs in poly-time if for every string s , $A(s)$ terminates in at most $p(|s|)$ "steps", where $p(\cdot)$ is some polynomial.

↑
length of s

PRIMES: $X = \{ 2, 3, 5, 7, 11, 13, 17, 23, 29, 31, 37, \dots \}$

Algorithm. [Agrawal-Kayal-Saxena, 2002] $p(|s|) = |s|^8$.

Definition of P

P. Decision problems for which there is a poly-time algorithm.

Problem	Description	Algorithm	Yes	No
MULTIPLE	Is x a multiple of y ?	Grade school division	51, 17	51, 16
RELPRIME	Are x and y relatively prime?	Euclid (300 BCE)	34, 39	34, 51
PRIMES	Is x prime?	AKS (2002)	53	51
EDIT-DISTANCE	Is the edit distance between x and y less than 5?	Dynamic programming	neither neither	acgggt ttttta
LSOLVE	Is there a vector x that satisfies $Ax = b$?	Gauss-Edmonds elimination	$\begin{bmatrix} 0 & 1 & -1 \\ 2 & 4 & -2 \\ 0 & 3 & 15 \end{bmatrix}, \begin{bmatrix} 4 \\ 2 \\ 36 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

Certification algorithm intuition.

- Certifier views things from "managerial" viewpoint.
- Certifier doesn't determine whether $s \in X$ on its own; rather, it checks a proposed proof t that $s \in X$.

Def. Algorithm $C(s, t)$ is a **certifier** for problem X if for every string s , $s \in X$ iff there exists a string t such that $C(s, t) = \text{yes}$.

"certificate" or "witness"

NP. Decision problems for which there exists a **poly-time certifier**.

$C(s, t)$ is a poly-time algorithm and $|t| \leq p(|s|)$ for some polynomial $p(\cdot)$.

Remark. NP stands for **nondeterministic polynomial-time**.

Certifiers and Certificates: Composite

COMPOSITES. Given an integer s , is s composite?

Certificate. A nontrivial factor t of s . Note that such a certificate exists iff s is composite. Moreover $|t| \leq |s|$.

Certifier.

```
boolean C(s, t) {
    if (t ≤ 1 or t ≥ s)
        return false
    else if (s is a multiple of t)
        return true
    else
        return false
}
```

Instance. $s = 437,669$.

Certificate. $t = 541$ or 809 . $\leftarrow 437,669 = 541 \times 809$

Conclusion. COMPOSITES is in NP.

Certifiers and Certificates: 3-Satisfiability

SAT. Given a CNF formula Φ , is there a satisfying assignment?

Certificate. An assignment of truth values to the n boolean variables.

Certifier. Check that each clause in Φ has at least one true literal.

Ex.

$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\overline{x_1} \vee \overline{x_3} \vee \overline{x_4})$$

instance s

$$x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$$

certificate t

Conclusion. SAT is in NP.

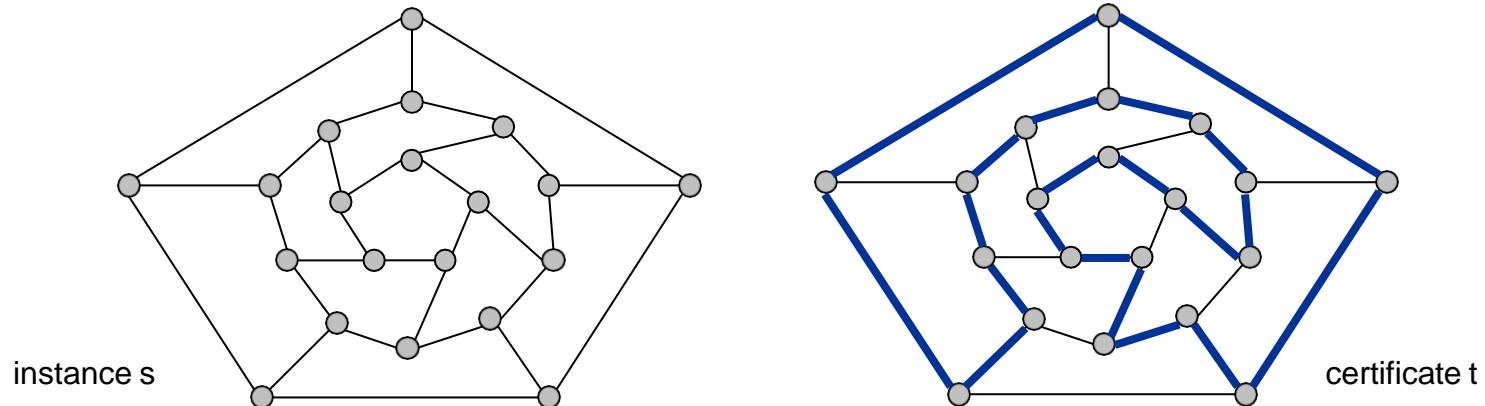
Certifiers and Certificates: Hamiltonian Cycle

HAM-CYCLE. Given an undirected graph $G = (V, E)$, does there exist a simple cycle C that visits every node?

Certificate. A permutation of the n nodes.

Certifier. Check that the permutation contains each node in V exactly once, and that there is an edge between each pair of adjacent nodes in the permutation.

Conclusion. HAM-CYCLE is in NP.



P, NP, EXP

P. Decision problems for which there is a **poly-time algorithm**.

EXP. Decision problems for which there is an **exponential-time algorithm**.

NP. Decision problems for which there is a **poly-time certifier**.

Claim. $P \subseteq NP$.

Pf. Consider any problem X in P .

- By definition, there exists a poly-time algorithm $A(s)$ that solves X .
- Certificate: $t = \varepsilon$, certifier $C(s, t) = A(s)$. ▀

Claim. $NP \subseteq EXP$.

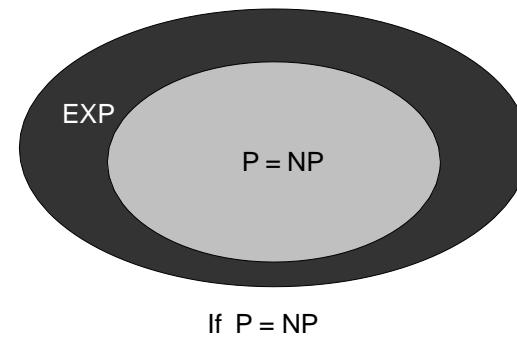
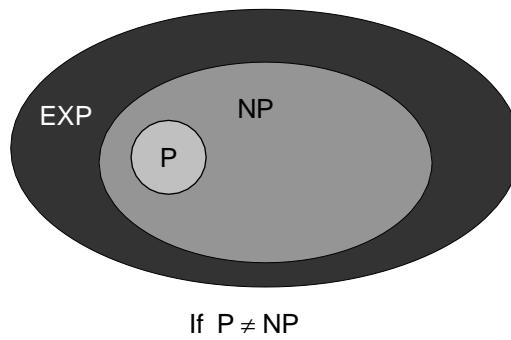
Pf. Consider any problem X in NP .

- By definition, there exists a poly-time certifier $C(s, t)$ for X .
- To solve input s , run $C(s, t)$ on all strings t with $|t| \leq p(|s|)$.
- Return yes , if $C(s, t)$ returns yes for any of these. ▀

The Main Question: P Versus NP

Does $P = NP$? [Cook 1971, Edmonds, Levin, Yablonski, Gödel]

- Is the decision problem as easy as the certification problem?
- Clay \$1 million prize.



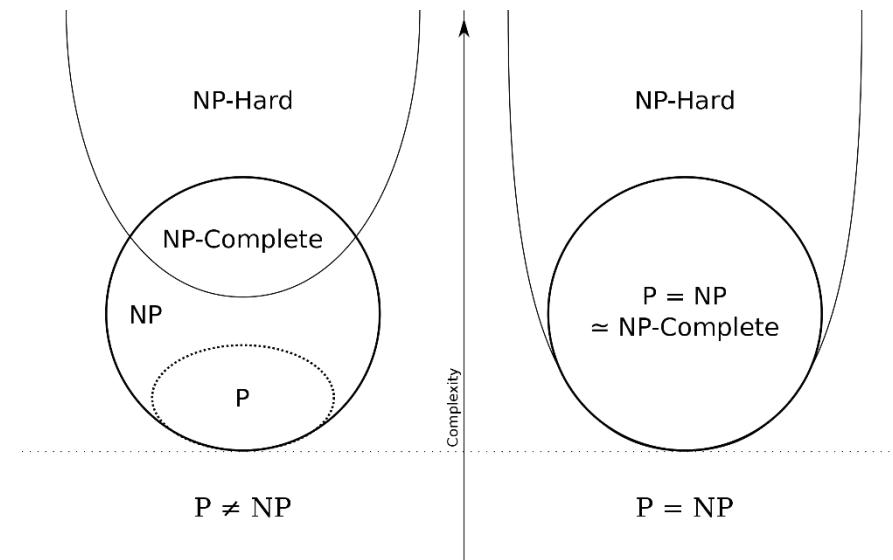
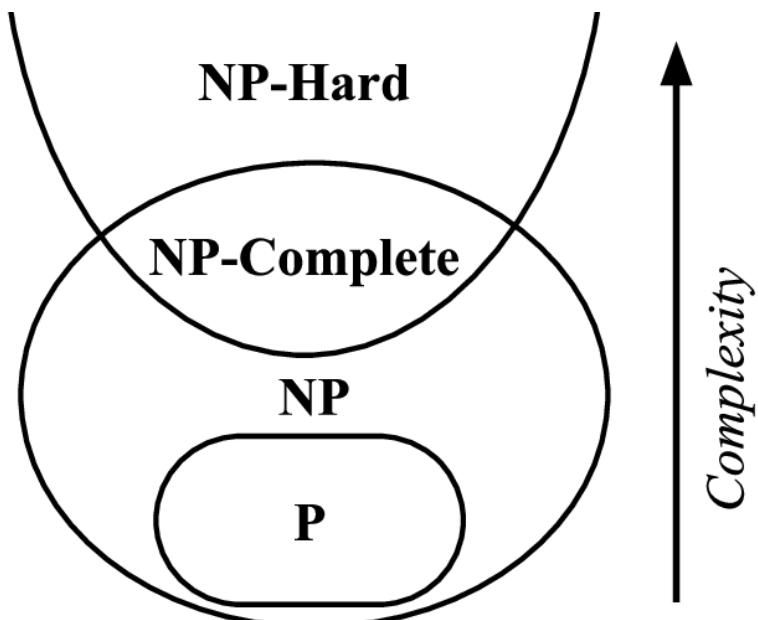
would break RSA cryptography
(and potentially collapse economy)

If yes: Efficient algorithms for 3-COLOR, TSP, FACTOR, SAT, ...

If no: No efficient algorithms possible for 3-COLOR, TSP, SAT, ...

Consensus opinion on $P = NP$? Probably no.

NP-Completeness



NP-Complete

NP-complete. A problem Y in NP with the property that for every problem X in NP, $X \leq_p Y$.

Theorem. Suppose Y is an NP-complete problem. Then Y is solvable in poly-time iff $P = NP$.

Pf. \Leftarrow If $P = NP$ then Y can be solved in poly-time since Y is in NP.

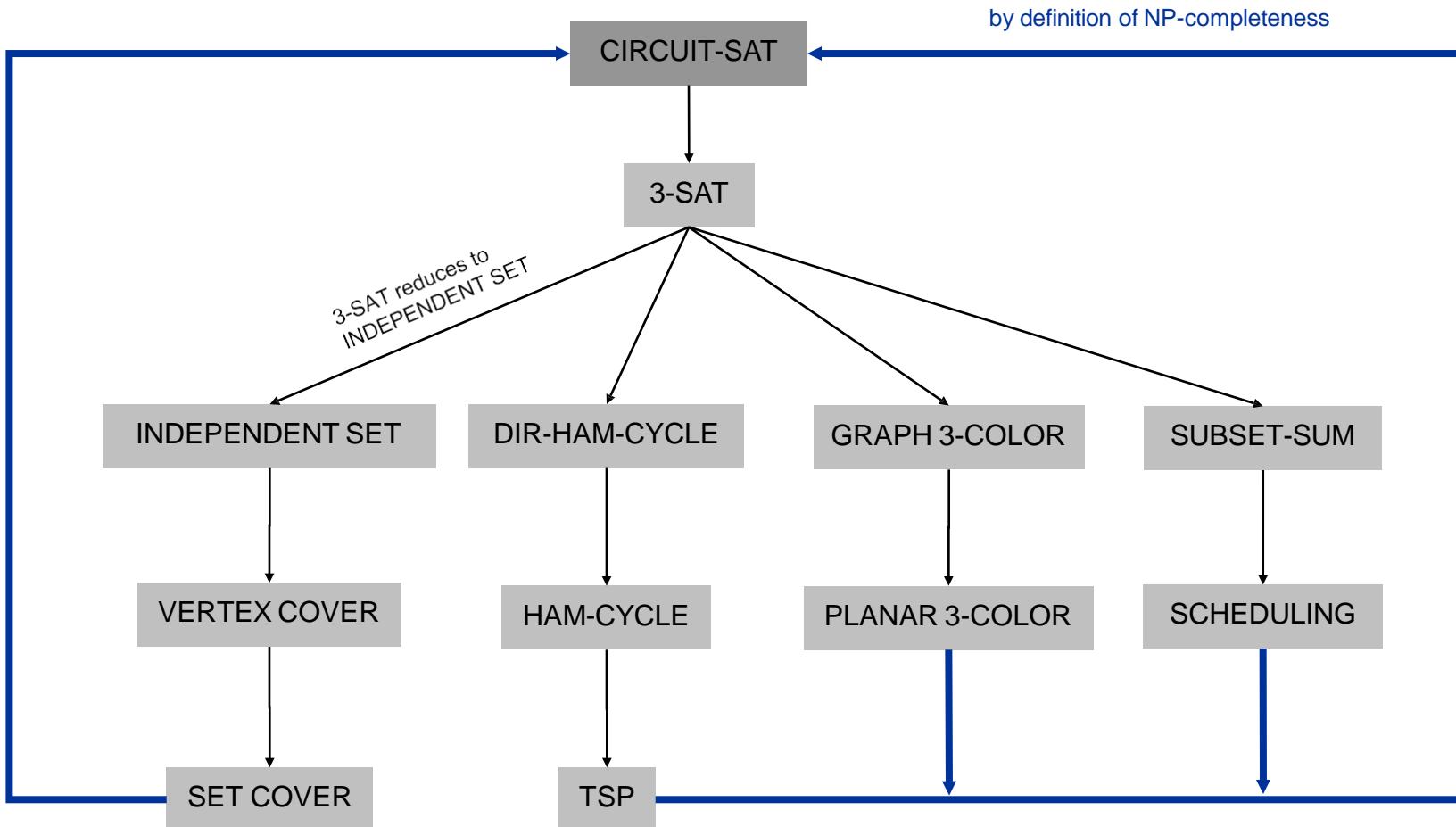
Pf. \Rightarrow Suppose Y can be solved in poly-time.

- Let X be any problem in NP. Since $X \leq_p Y$, we can solve X in poly-time. This implies $NP \subseteq P$.
- We already know $P \subseteq NP$. Thus $P = NP$. ■

Fundamental question. Do there exist "natural" NP-complete problems?

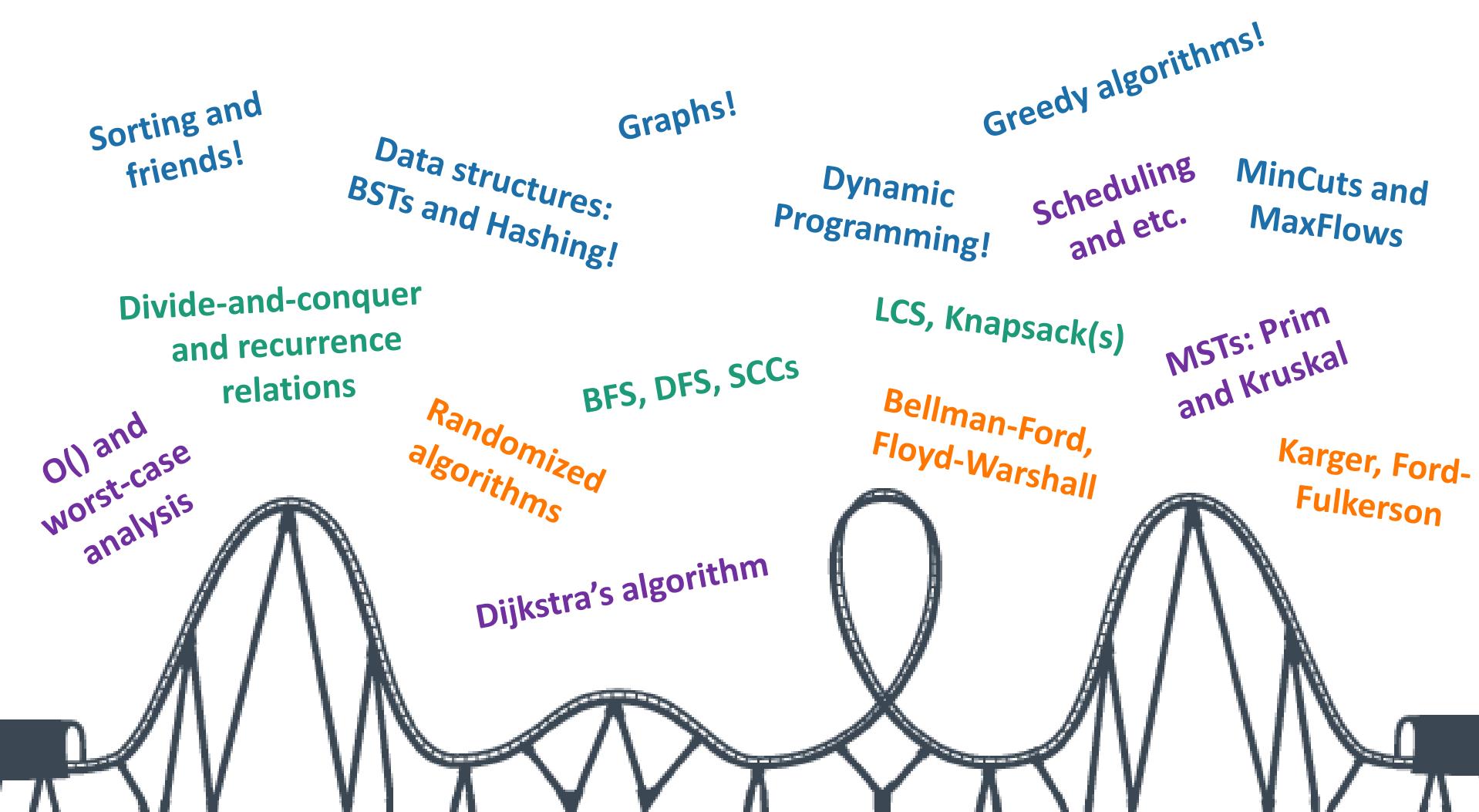
NP-Completeness

Observation. All problems below are NP-complete and polynomial reduce to one another!



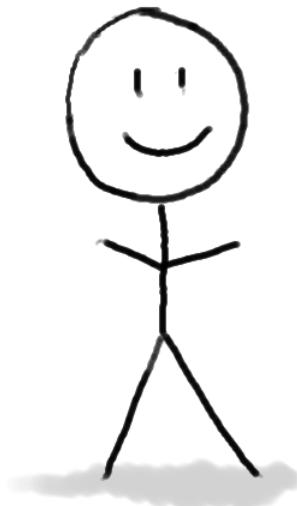
REVIEW

It's been a fun ride...



General approach to algorithm design and analysis

Can I do better?



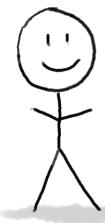
Algorithm designer

To answer this question we need
both **rigor** and **intuition**:

Detail-oriented
Precise
Rigorous

Big-picture
Intuitive
Hand-wavey

We needed more details



Does it work?
Is it fast?

What
does that
mean??



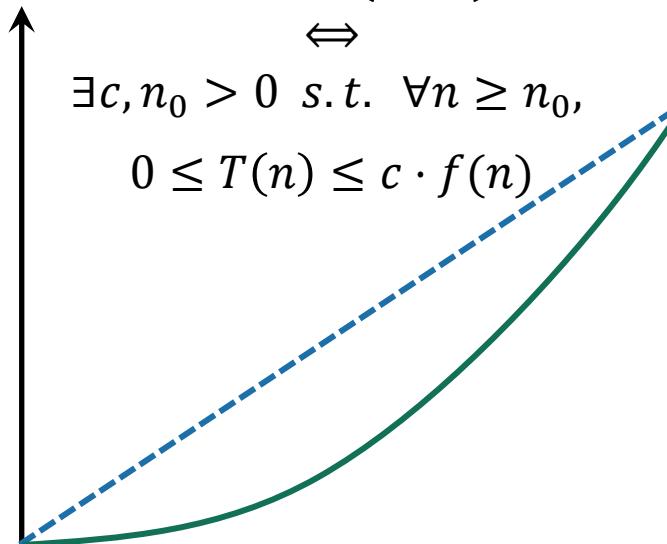
Worst-case analysis



HERE IS AN
INPUT!

big-Oh notation

$$T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq T(n) \leq c \cdot f(n)$$



Algorithm design paradigm: divide and conquer

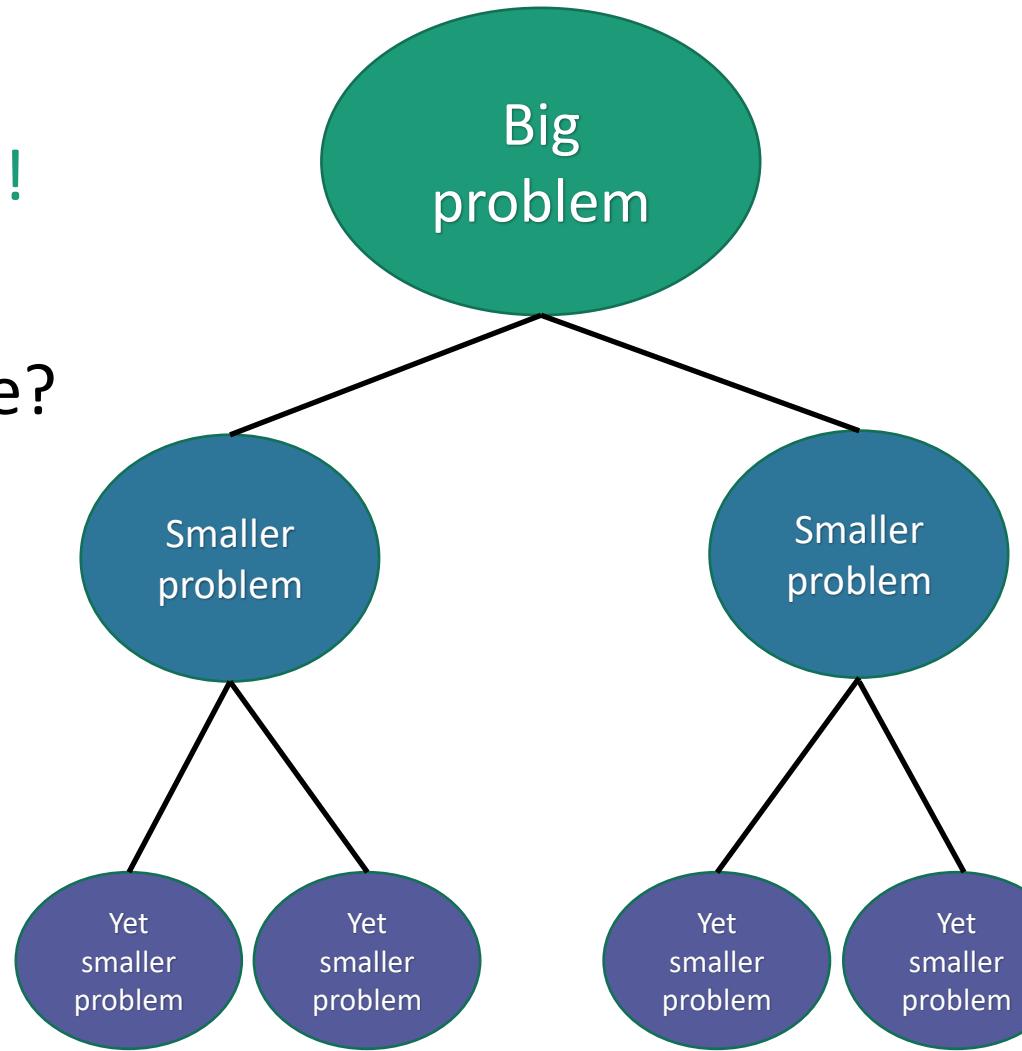
- Like MergeSort!
- Or Karatsuba's algorithm!
- Or SELECT!
- How do we analyze these?

By careful analysis!

Useful shortcut, the **master method** is.

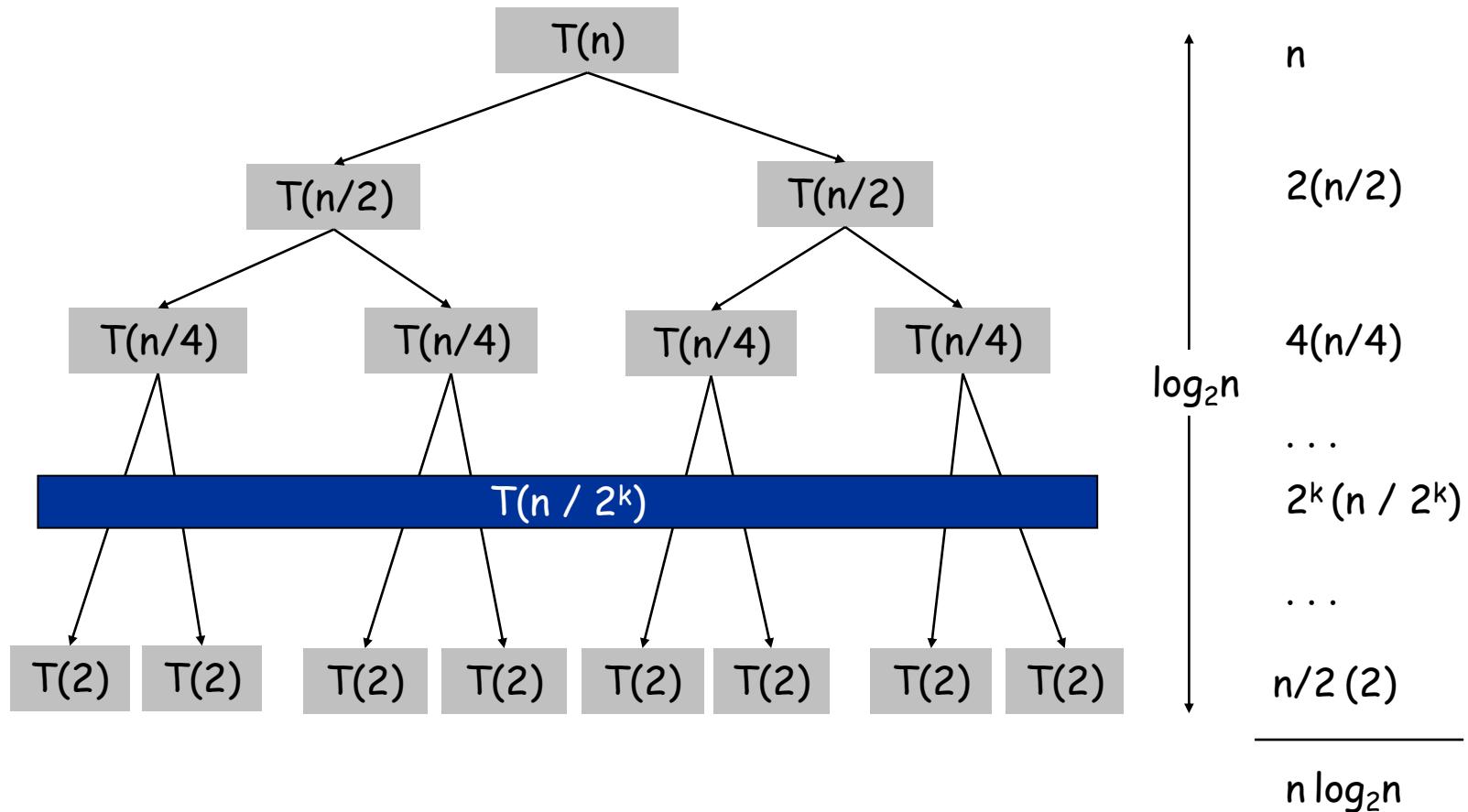


Jedi master Yoda



Recap- Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Recap-Why Recurrences?

- The complexity of many interesting algorithms is easily expressed as a recurrence – especially divide and conquer algorithms
- The form of the algorithm often yields the form of the recurrence
- The complexity of recursive algorithms is readily expressed as a recurrence.

Recap-Why solve recurrences?

- To make it easier to compare the complexity of two algorithms
- To make it easier to compare the complexity of the algorithm to standard reference functions.

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d).$$

Recap-Master Theorem

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

- Needlessly recursive integer mult.

- $T(n) = 4 T(n/2) + O(n)$
- $T(n) = O(n^2)$

$$\begin{array}{l} a = 4 \\ b = 2 \\ d = 1 \end{array} \quad a > b^d$$



- Karatsuba integer multiplication

- $T(n) = 3 T(n/2) + O(n)$
- $T(n) = O(n^{\log_2(3)} \approx n^{1.6})$

$$\begin{array}{l} a = 3 \\ b = 2 \\ d = 1 \end{array} \quad a > b^d$$



- MergeSort

- $T(n) = 2T(n/2) + O(n)$
- $T(n) = O(n \log(n))$

$$\begin{array}{l} a = 2 \\ b = 2 \\ d = 1 \end{array} \quad a = b^d$$



- That other one

- $T(n) = T(n/2) + O(n)$
- $T(n) = O(n)$

$$\begin{array}{l} a = 1 \\ b = 2 \\ d = 1 \end{array} \quad a < b^d$$



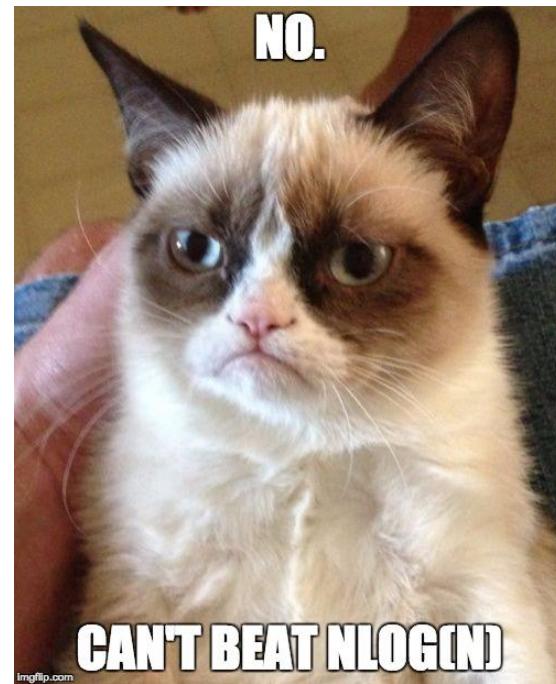
While we're on the topic of sorting Why not use randomness?

- We analyzed **QuickSort!**
- Still **worst-case input**, but we use **randomness** after the input is chosen.
- Always correct, usually fast.
 - This is a Las Vegas algorithm

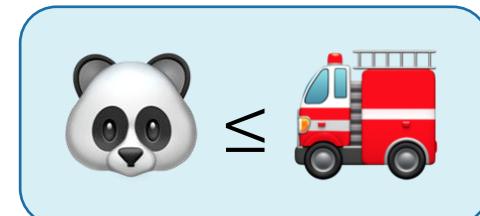


All this sorting is making me wonder... Can we do better?

- Depends on who you ask:



- **RadixSort** takes time $O(n)$ if the objects are, for example, small integers!
- Can't do better in a **comparison-based** model.



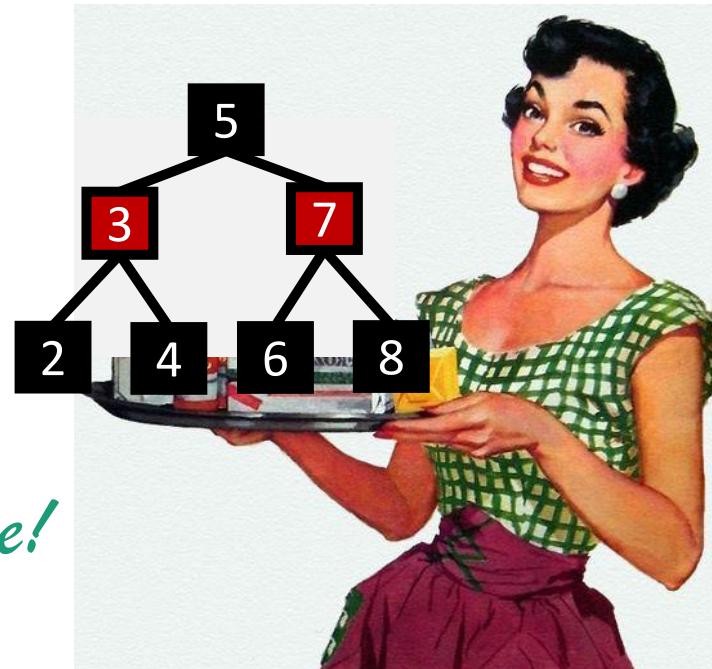
beyond sorted arrays/linked lists: Binary Search Trees!

- Useful data structure!
- Especially the self-balancing ones!

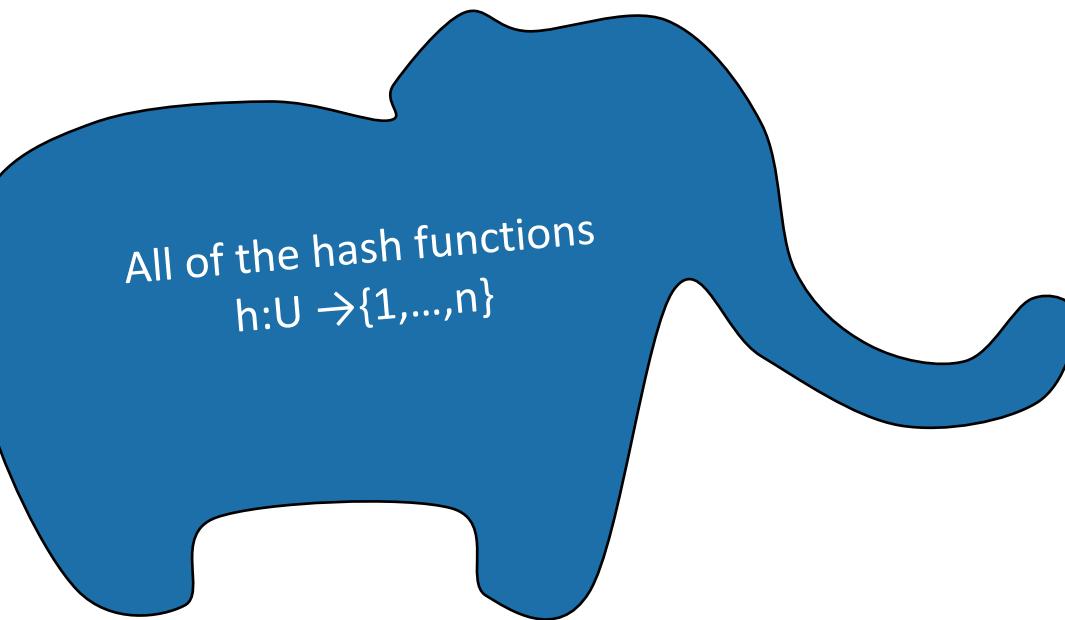
Red-Black tree!

Maintain balance by stipulating that
black nodes are balanced, and
that there aren't too many **red
nodes**.

It's just good sense!



Another way to store things Hash tables!



Choose h randomly from a universal hash family.



It's better if the hash family is small!
Then it takes less space to store h .

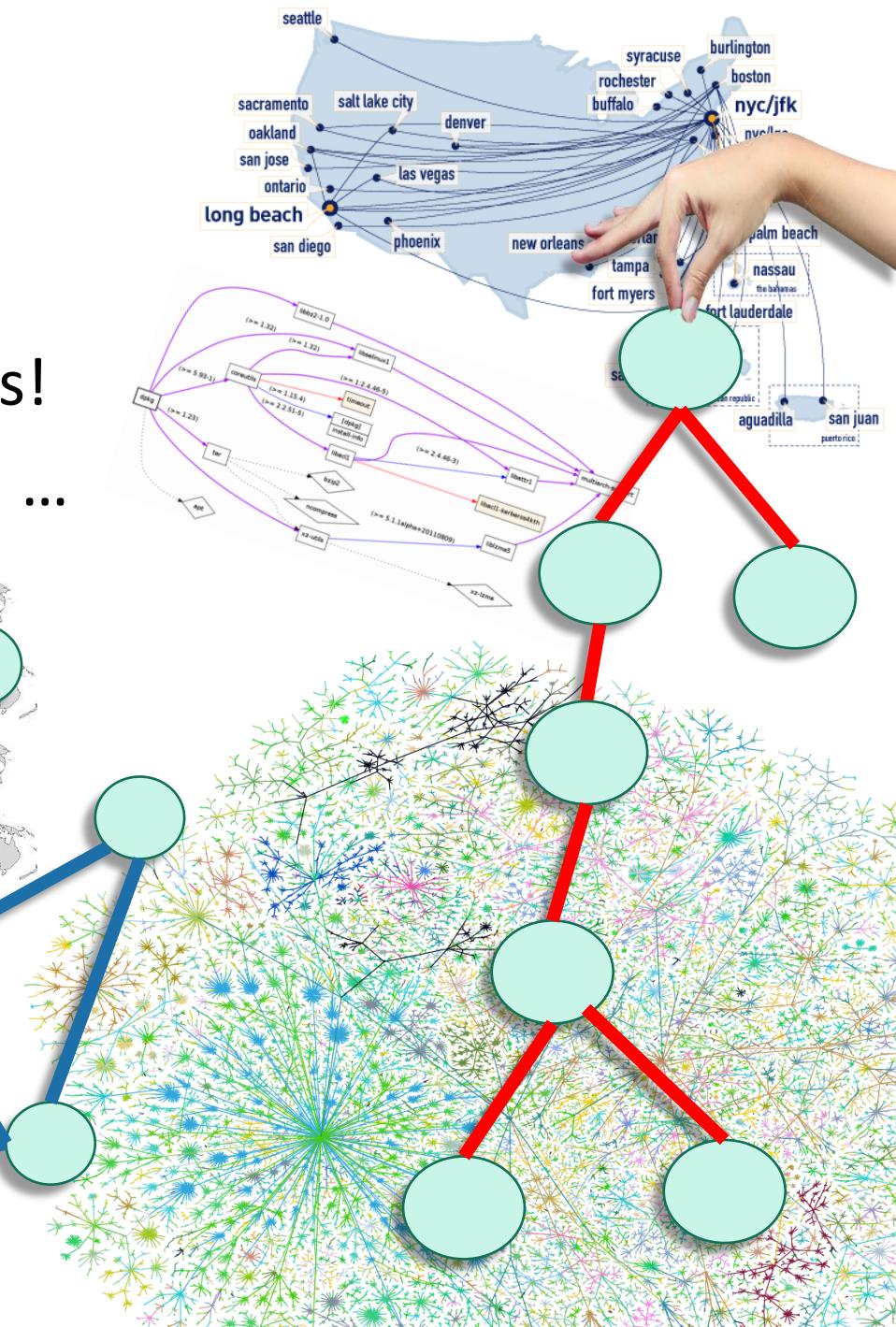
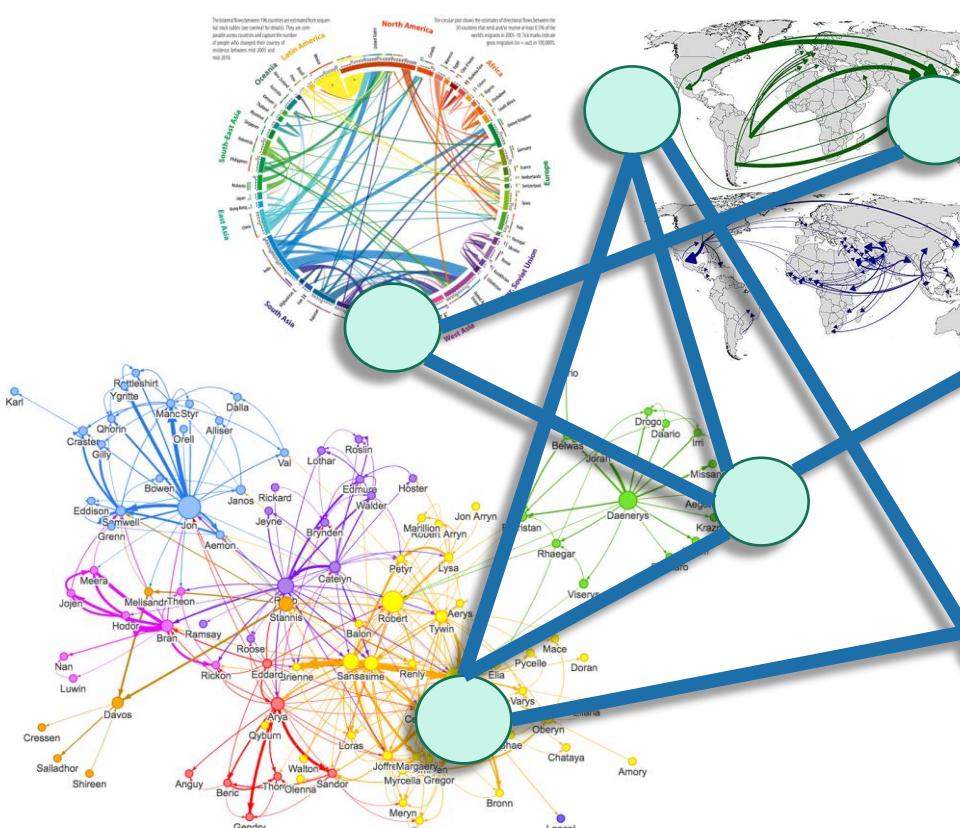


hash function h



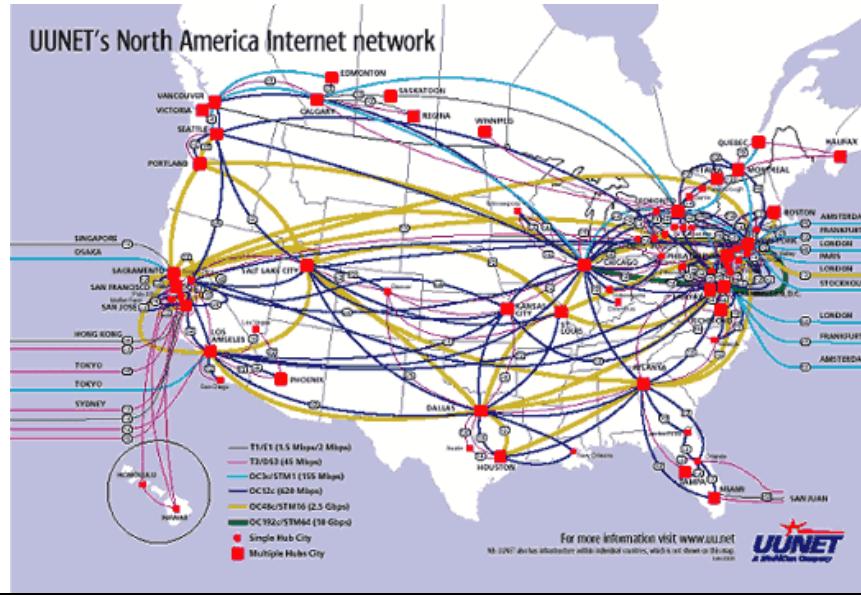
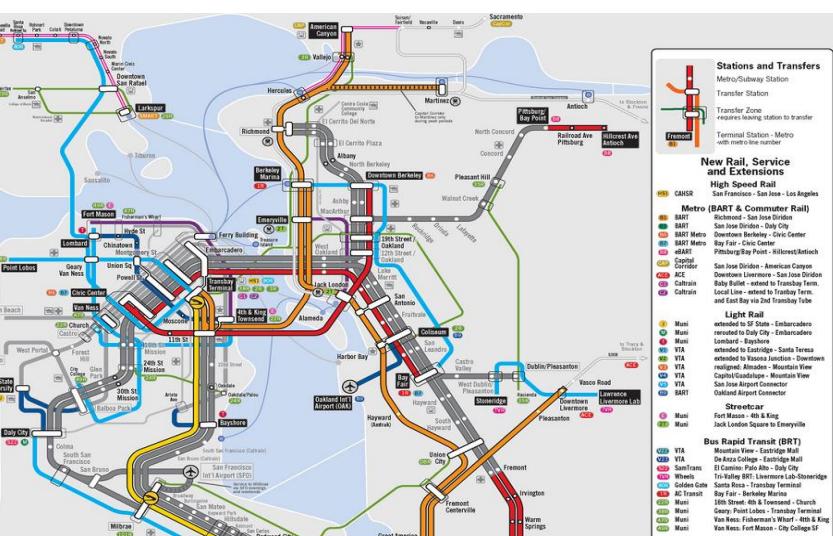
OMG GRAPHS

- BFS, DFS, and applications!
- SCCs, Topological sorting, ...



A fundamental graph problem: shortest paths

- Eg, transit planning, packet routing, ...
 - Dijkstra!



```
DN0a22a0e3:~ mary$ traceroute -a www.ethz.ch
traceroute to www.ethz.ch (129.132.19.216), 64 hops max, 52 byte packets
 1 [AS0] 10.34.160.2 (10.34.160.2) 38.168 ms 31.272 ms 28.841 ms
 2 [AS0] cwa-vrtr.sunet (10.21.21.196.28) 33.769 ms 28.245 ms 24.373 ms
 3 [AS32] 171.66.2.229 (171.66.2.229) 24.468 ms 20.115 ms 23.223 ms
 4 [AS32] hpr-svl-rtr-vlan8.sunet (171.64.255.235) 24.644 ms 24.962 ms 17.453 ms
 5 [AS2152] hpr-svl-hpr2--stan-ge.cenic.net (137.164.27.161) 22.129 ms 4.902 ms 3.642 ms
 6 [AS2152] hpr-lax-hpr3--svl-hpr3-100ge.cenic.net (137.164.25.73) 12.125 ms 43.361 ms 32.3
 7 [AS2152] hpr-i2--lax-hpr2-r&e.cenic.net (137.164.26.201) 40.174 ms 38.399 ms 34.499 ms
 8 [AS0] et-4-0-0.4079.sdn-sw.lasv.net.internet2.edu (162.252.70.28) 46.573 ms 23.926 ms 17
 9 [AS0] et-5-1-0.4079.rtsw.salt.net.internet2.edu (162.252.70.31) 30.424 ms 25.770 ms 23.1
10 [AS0] et-4-0-0.4079.sdn-sw.denv.net.internet2.edu (162.252.70.8) 47.454 ms 57.273 ms 73.
11 [AS0] et-4-1-0.4079.rtsw.kans.net.internet2.edu (162.252.70.11) 70.825 ms 67.809 ms 62.1
12 [AS0] et-4-1-0.4070.rtsw.chic.net.internet2.edu (198.71.47.206) 77.937 ms 57.421 ms 63.6
13 [AS0] et-0-1-0.4079.sdn-sw.ashb.net.internet2.edu (162.252.70.60) 77.682 ms 71.993 ms 73
14 [AS0] et-4-1-0.4079.rtsw.wash.net.internet2.edu (162.252.70.65) 71.565 ms 74.988 ms 71.0
15 [AS21320] internet2-gw.mx1.lon.uk.geant.net (62.40.124.44) 154.926 ms 145.606 ms 145.872
16 [AS21320] ae0.mx1.lon2.uk.geant.net (62.40.98.79) 146.565 ms 146.604 ms 146.801 ms
17 [AS21320] ae0.mx1.par.fr.geant.net (62.40.98.77) 153.289 ms 184.995 ms 152.682 ms
18 [AS21320] ae2.mx1.gen.ch.geant.net (62.40.98.153) 160.283 ms 160.104 ms 164.147 ms
19 [AS21320] swice1-100ge-0-3-0-1.switch.ch (62.40.124.22) 162.068 ms 160.595 ms 163.095 ms
20 [AS559] swizh1-100ge-0-1-0-1.switch.ch (130.59.36.94) 165.824 ms 164.216 ms 163.983 ms
21 [AS559] swiez3-100ge-0-1-0-4.switch.ch (130.59.38.109) 164.269 ms 164.370 ms 163.929 ms
22 [AS559] rou-gw-lee-tengig-to-switch.ethz.ch (192.33.92.1) 164.082 ms 170.645 ms 165.372
23 [AS559] rou-fw-rz-rz-gw.ethz.ch (192.33.92.169) 164.773 ms 165.193 ms 172.158 ms
```

Bellman-Ford and Floyd-Warshall
were examples of...

Dynamic Programming!

- Not programming in an action movie.

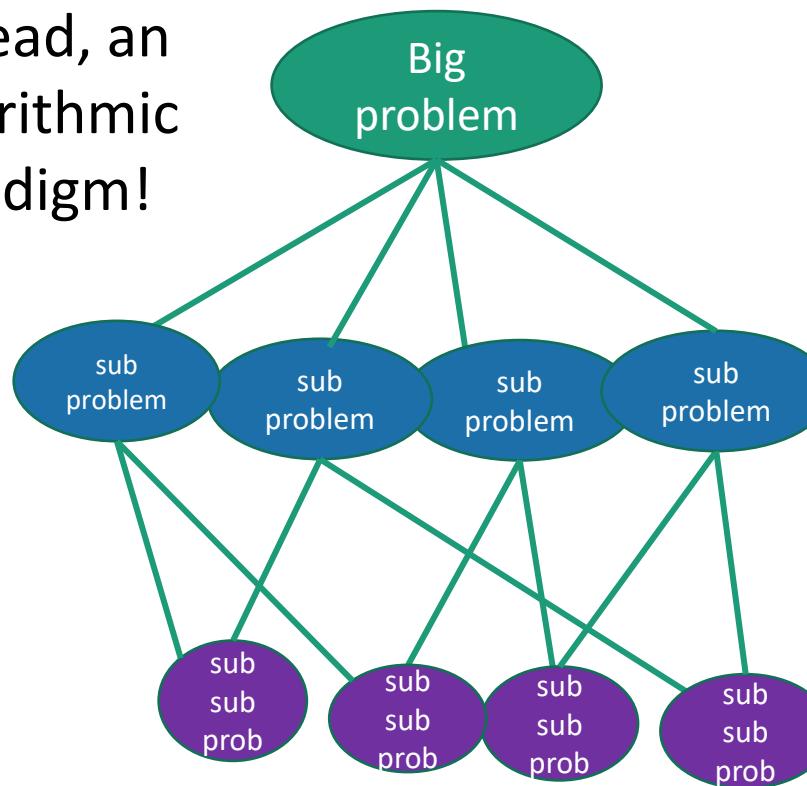


- Step 1:** Identify optimal substructure.
- Step 2:** Find a recursive formulation for the value of the optimal solution.
- Steps 3-5:** Use dynamic programming:
fill in a table to find the answer!

Instead, an
algorithmic
paradigm!



We saw many other examples, including Longest Common Subsequence and Knapsack Problems.



Dynamic Programming!

- Dynamic programming is an **algorithm design paradigm**.
- Basic idea:
 - Identify **optimal sub-structure**
 - Optimum to the big problem is built out of optima of small sub-problems
 - Take advantage of **overlapping sub-problems**
 - Only solve each sub-problem once, then use it again and again
 - Keep track of the solutions to sub-problems in a table as you build to the final solution.

Dynamic Programming-Recap

- We saw examples of how to come up with dynamic programming algorithms.
 - Longest Common Subsequence
 - Knapsack two ways
 - (If time) maximal independent set in trees.
- There is a **recipe** for dynamic programming algorithms.

Recipe for 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.

Knapsack problem

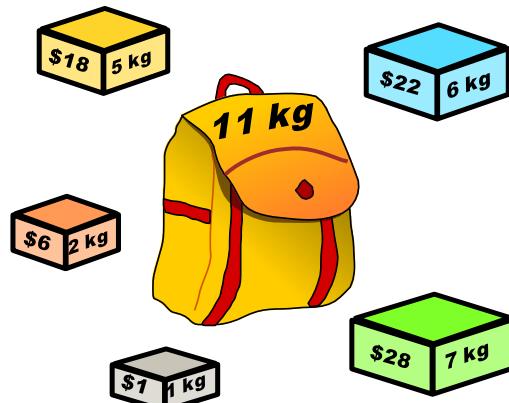
Goal. Pack knapsack so as to maximize total value of items taken.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$.
- Value of a subset of items = sum of values of individual items.
- Knapsack has weight limit of W .

Ex. The subset { 1, 2, 5 } has value \$35 (and weight 10).

Ex. The subset { 3, 4 } has value \$40 (and weight 11).

Assumption. All values and weights are integral.



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

weights and values
can be arbitrary
positive integers

Dynamic programming: two variables

Def. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \dots, i$, subject to weight limit w .

Goal. $OPT(n, W)$.

Case 1. $OPT(i, w)$ does not select item i .

- $OPT(i, w)$ selects best of $\{1, 2, \dots, i-1\}$ subject to weight limit w .

possibly because $w_i > w$

Case 2. $OPT(i, w)$ selects item i .

- Collect value v_i .
- New weight limit $= w - w_i$.
- $OPT(i, w)$ selects best of $\{1, 2, \dots, i-1\}$ subject to new weight limit.

optimal substructure property
(proof via exchange argument)

Bellman equation.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max \{ OPT(i-1, w), v_i + OPT(i-1, w - w_i) \} & \text{otherwise} \end{cases}$$

Sometimes we can take even better advantage of optimal substructure...with

Greedy algorithms

- Make a series of choices, and commit!

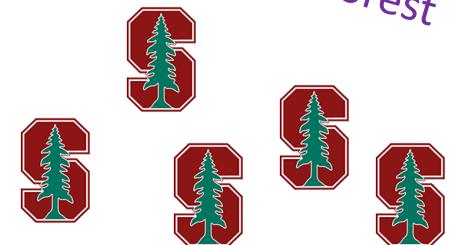


- Intuitively we want to show that our greedy choices never rule out success.
- Rigorously, we usually analyzed these by induction.
- Examples!
 - Activity Selection
 - Job Scheduling
 - Huffman Coding
 - Minimum Spanning Trees

*Prim's algorithm:
greedily grow a tree*

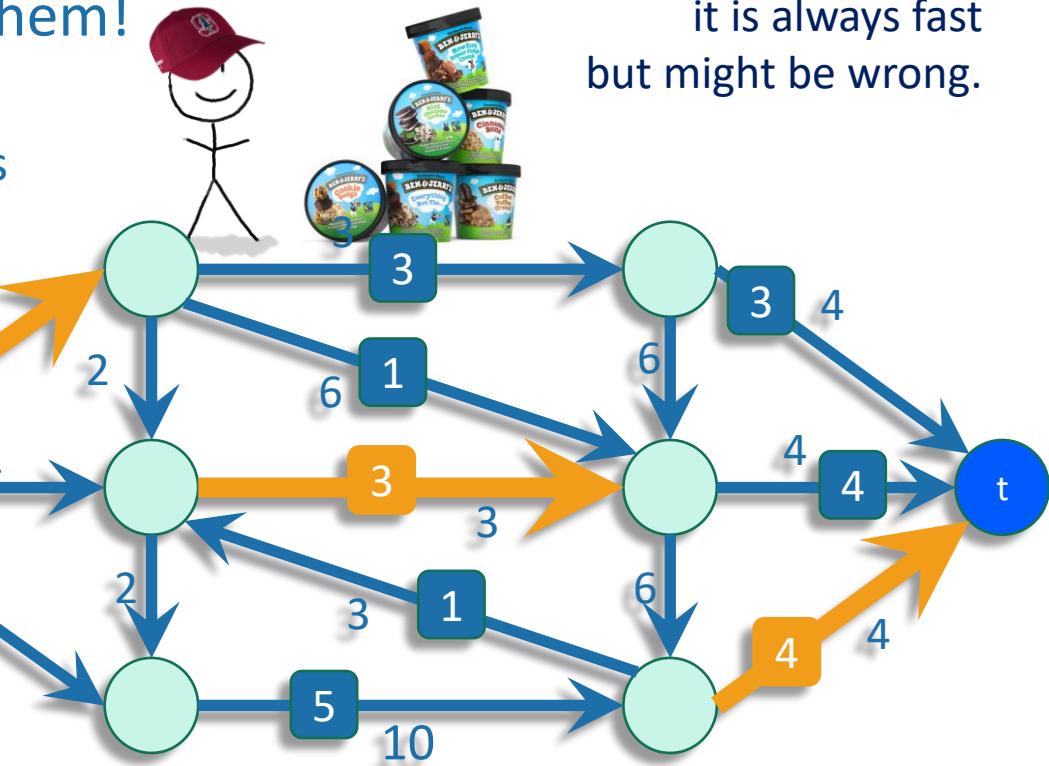
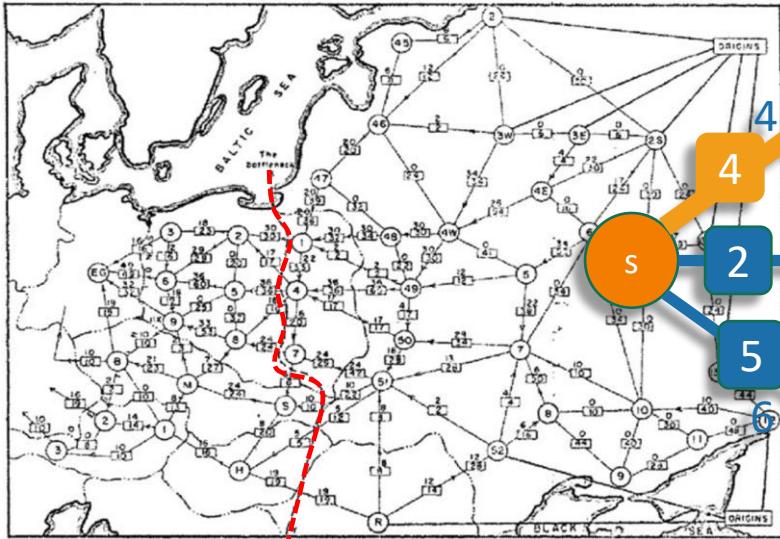


*Kruskal's algorithm:
greedily grow a forest*



Cuts and flows

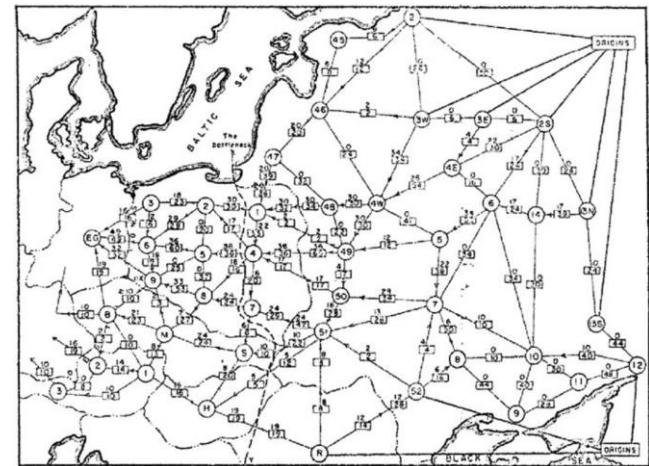
- Global minimum cut:
 - Karger's algorithm!
- minimum s-t cut:
 - is the same as maximum s-t flow!
 - Ford-Fulkerson can find them!
 - useful for routing
 - also assignment problems



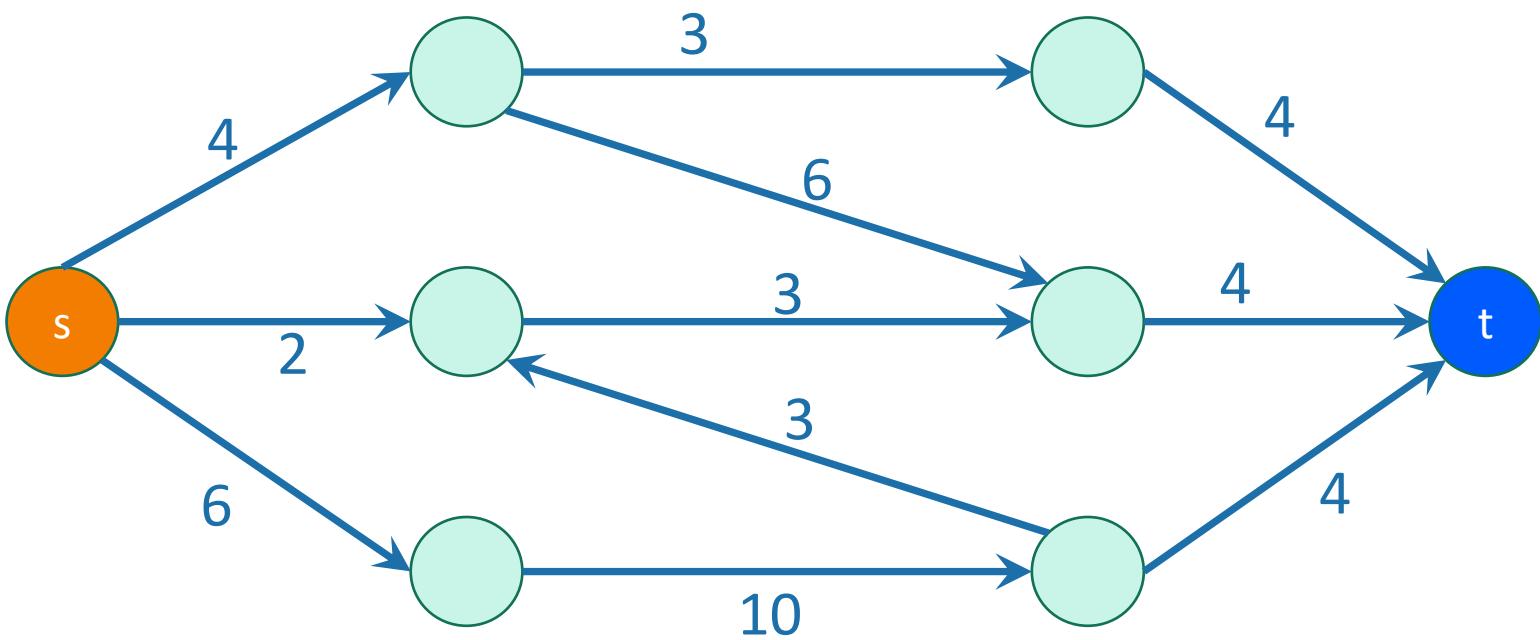
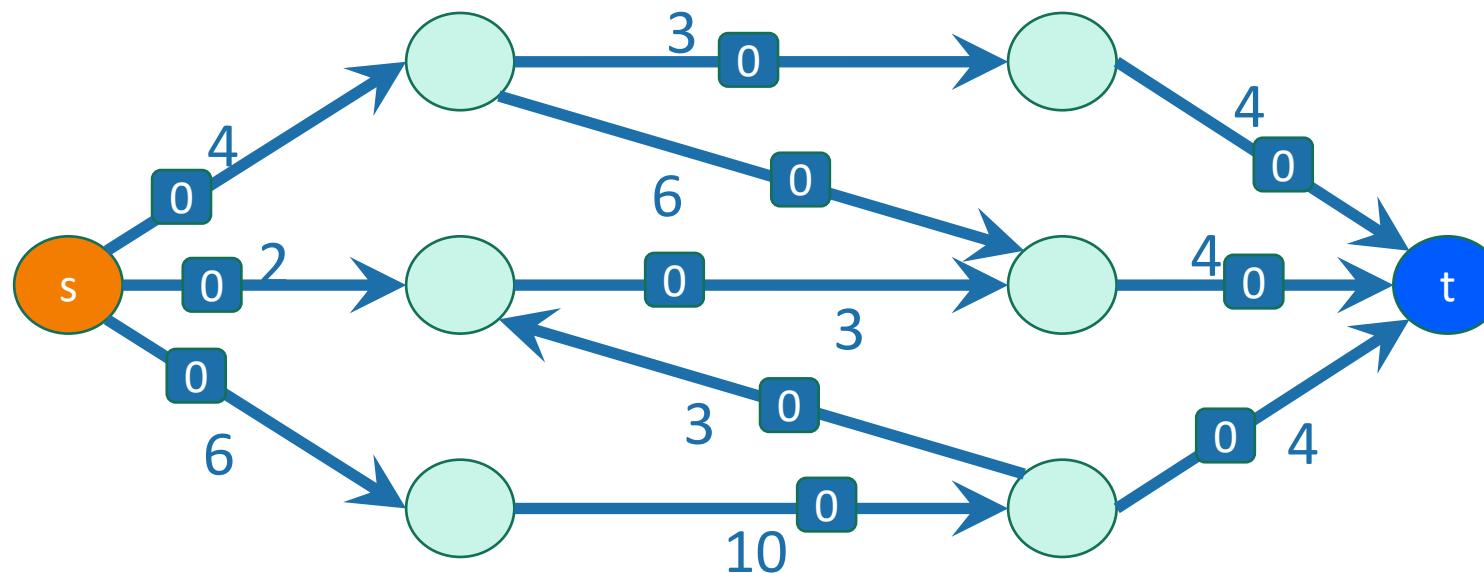
Karger's algorithm is a Monte-Carlo algorithm:
it is always fast
but might be wrong.

What have we learned?

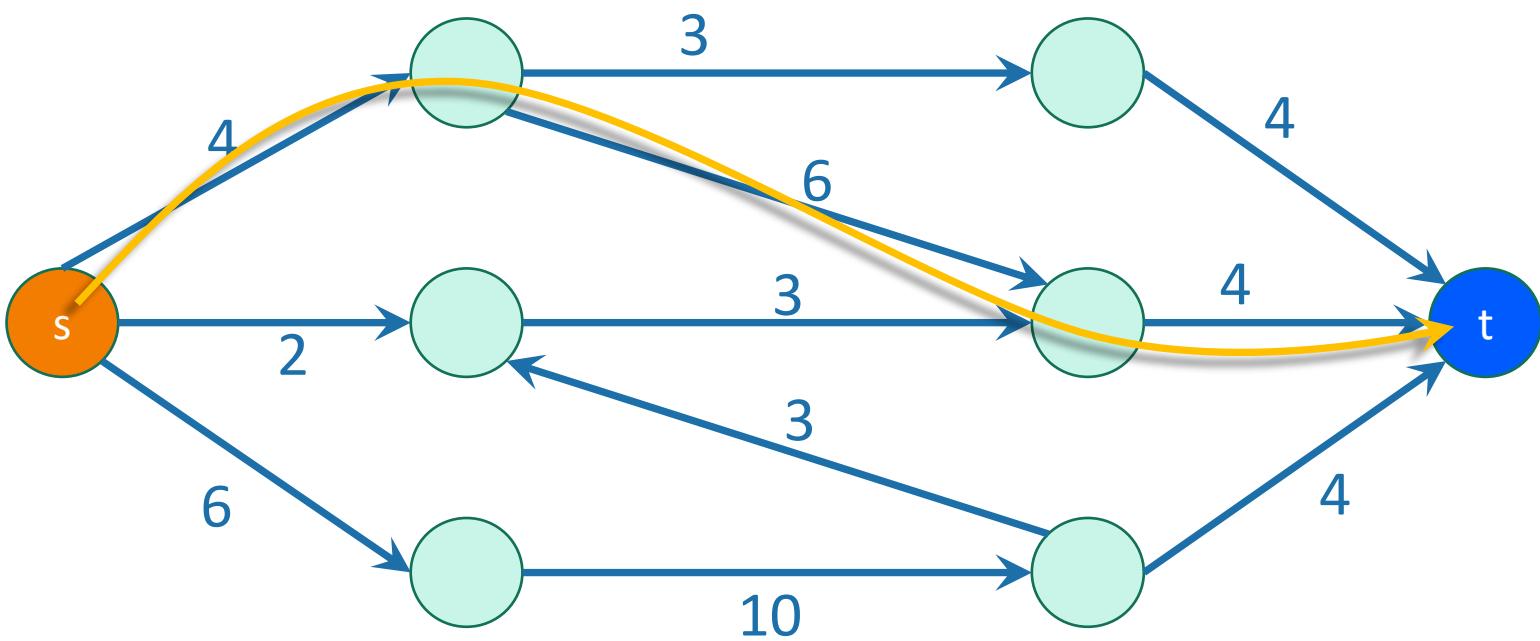
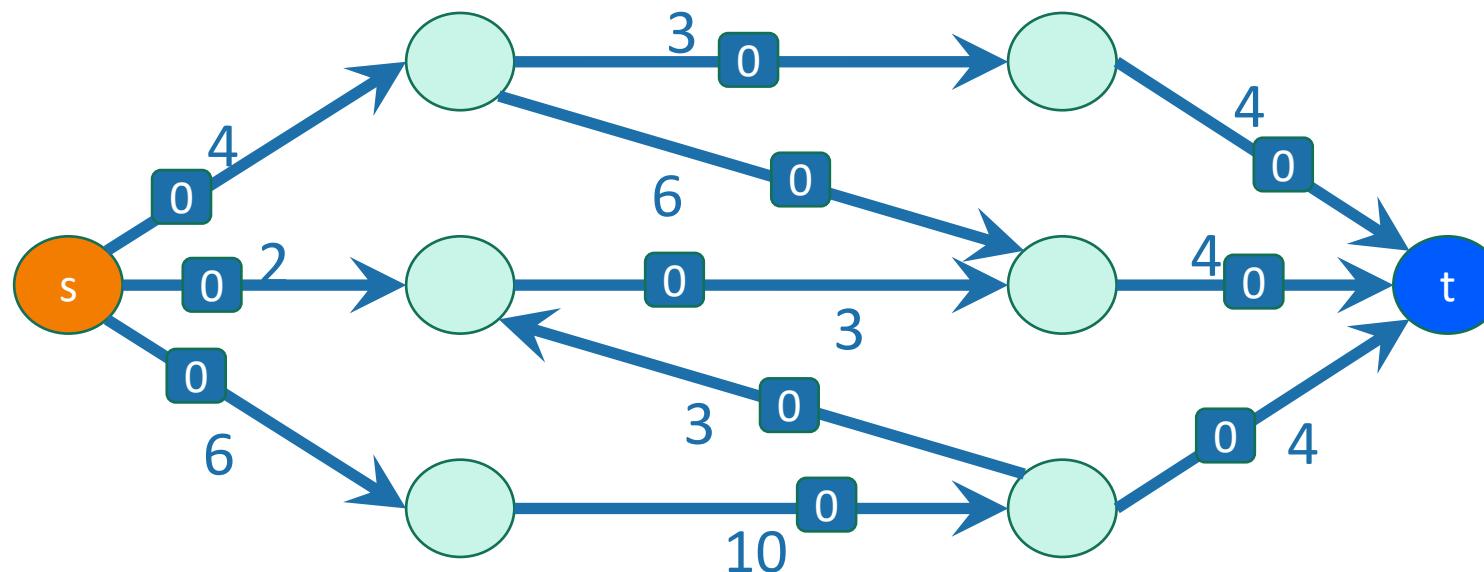
- Max s-t flow is equal to min s-t cut!
 - The USSR and the USA were trying to solve the same problem...
 - The Ford-Fulkerson algorithm can find the min-cut/max-flow.
 - Repeatedly improve your flow along an augmenting path.
 - **How long does this take???**



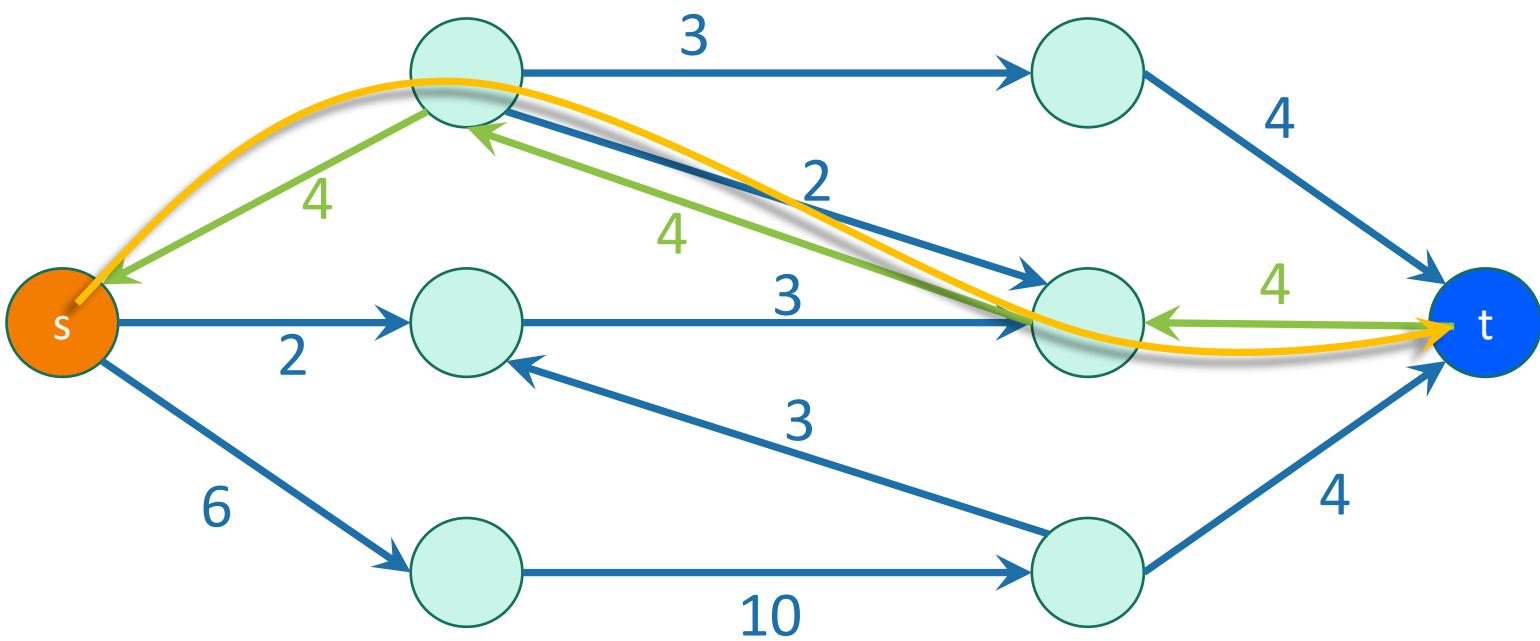
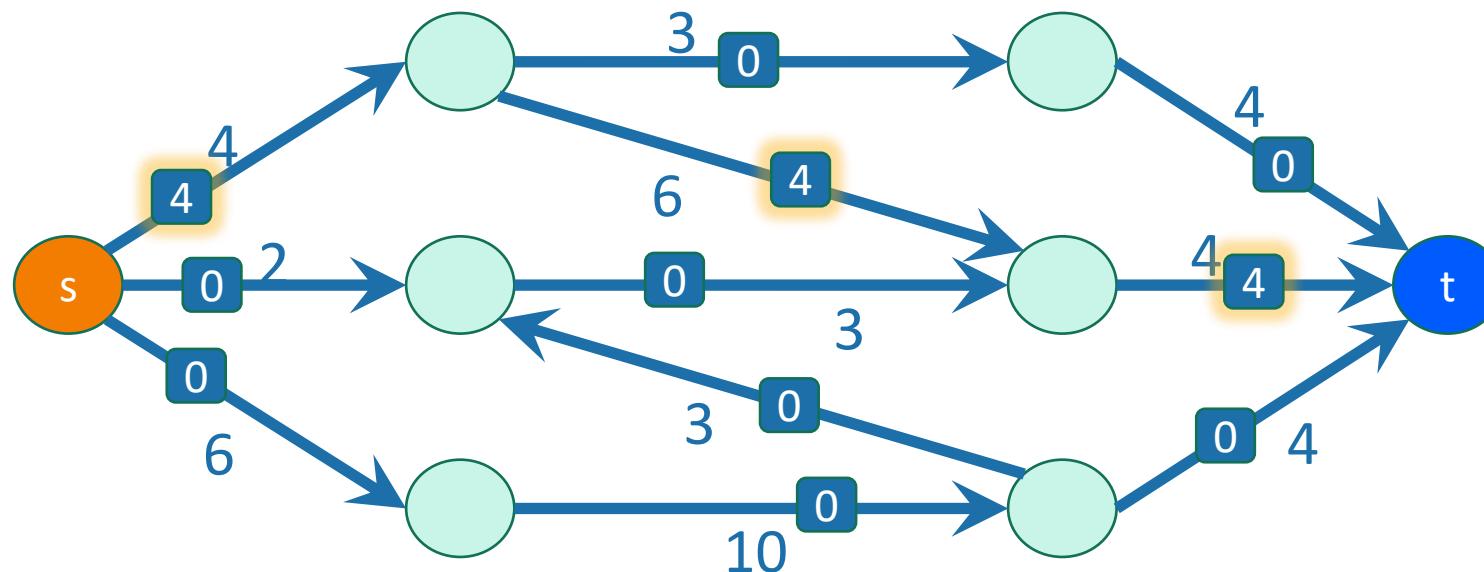
Example of Ford-Fulkerson



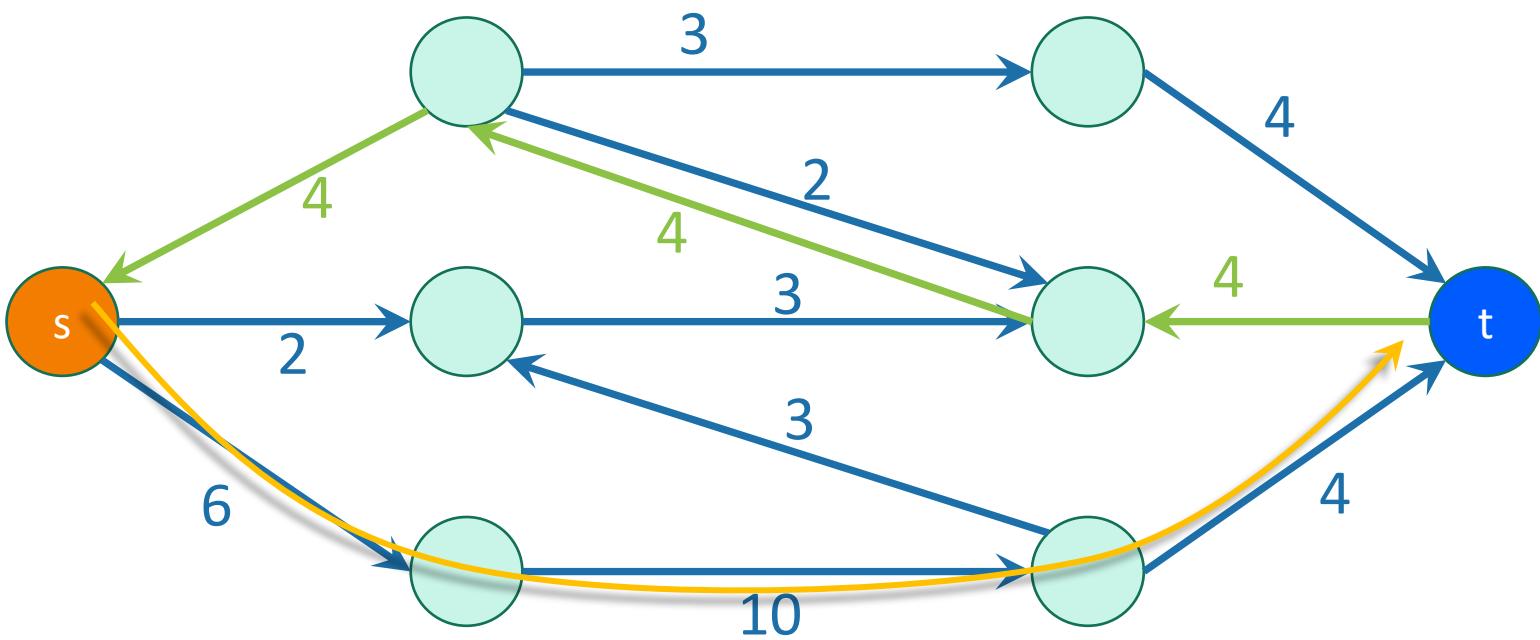
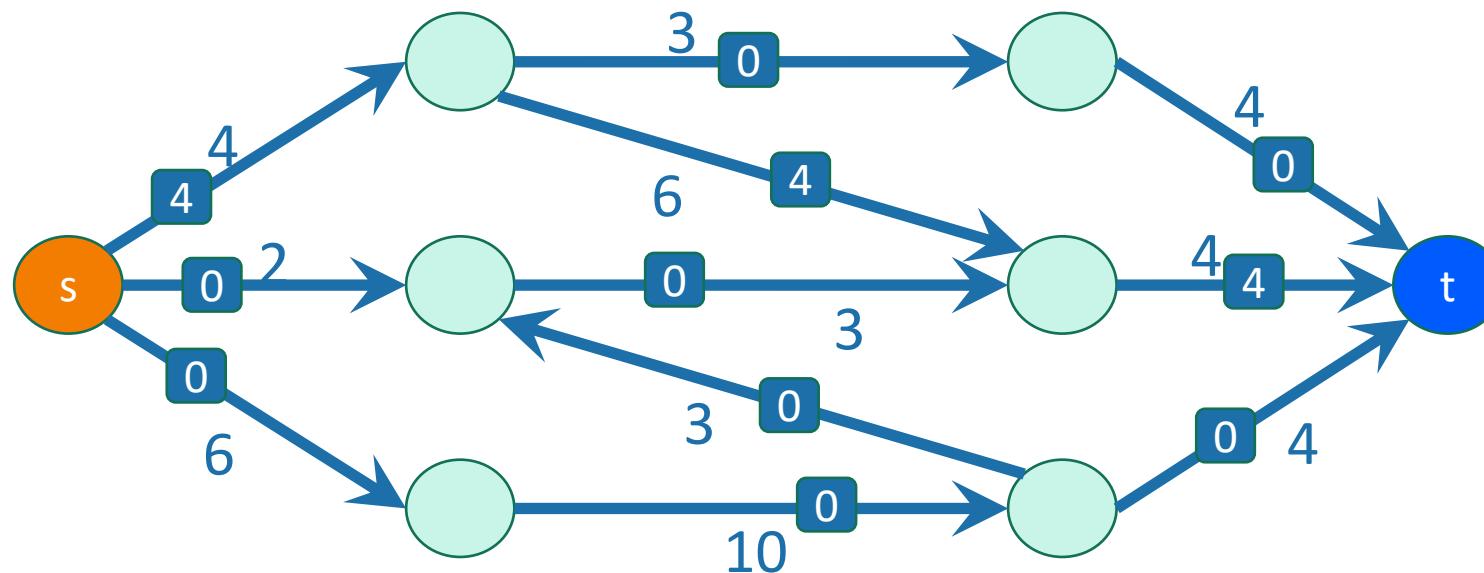
Example of Ford-Fulkerson



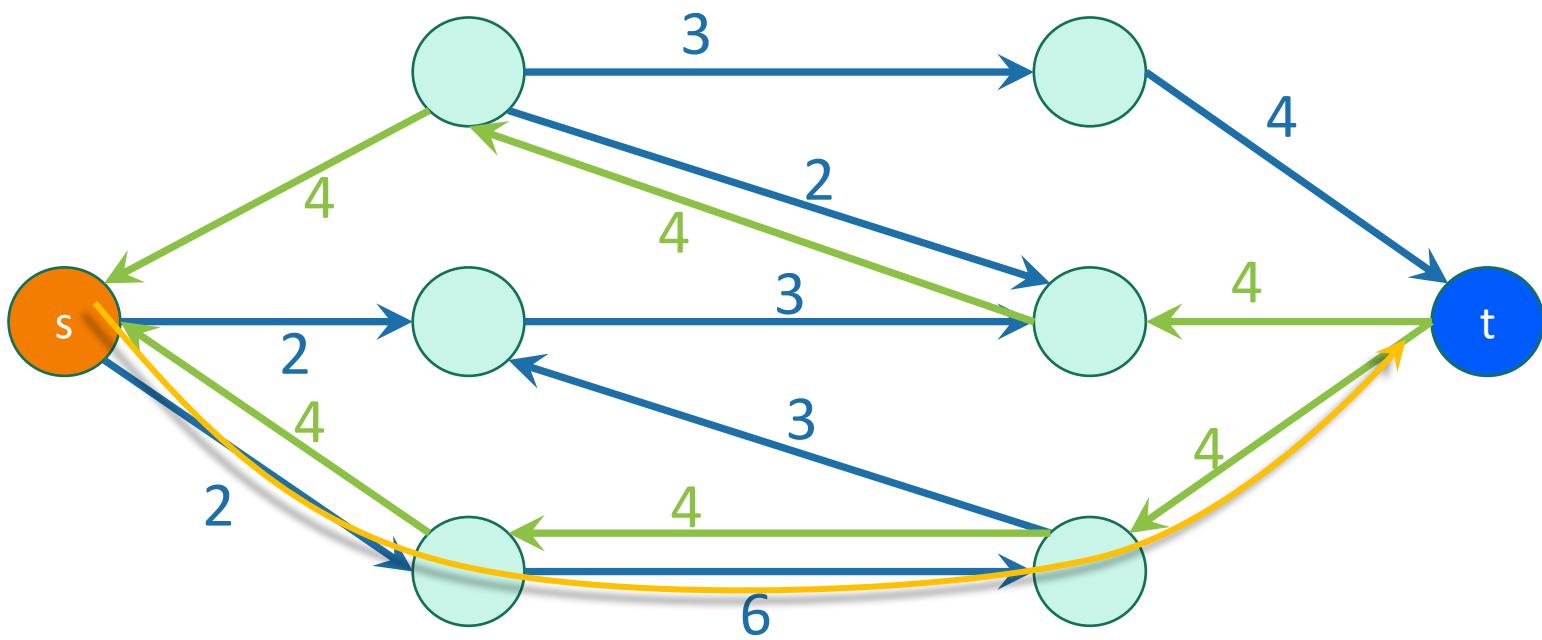
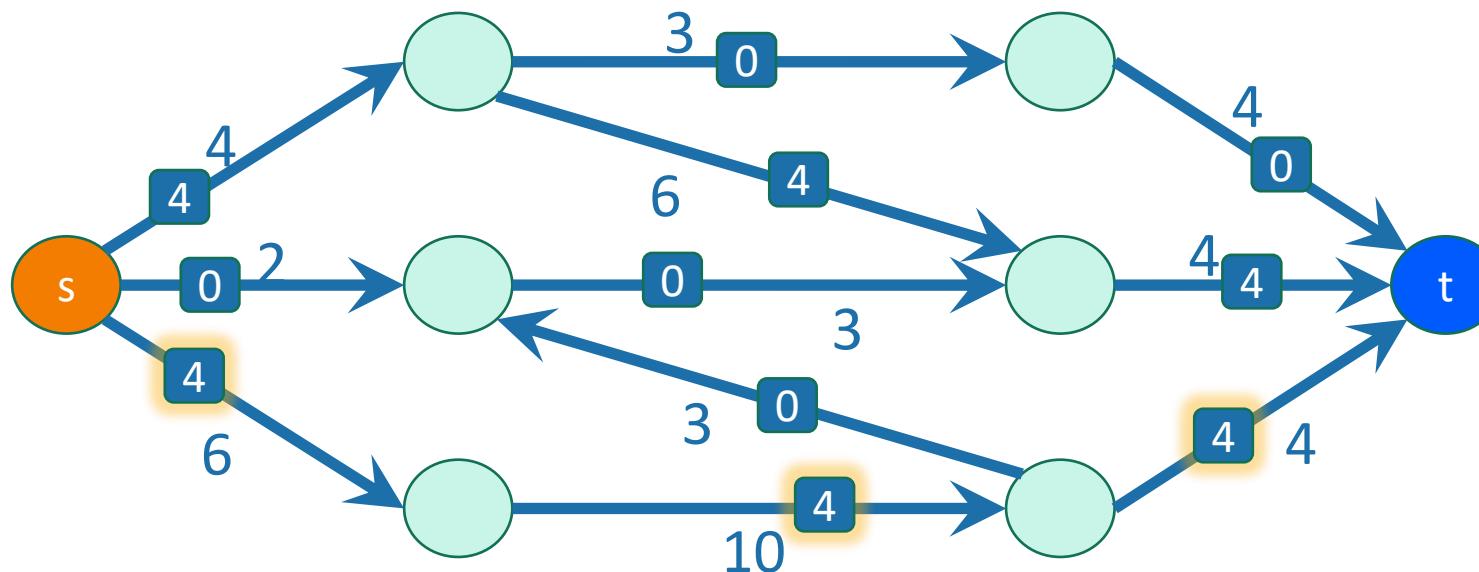
Example of Ford-Fulkerson



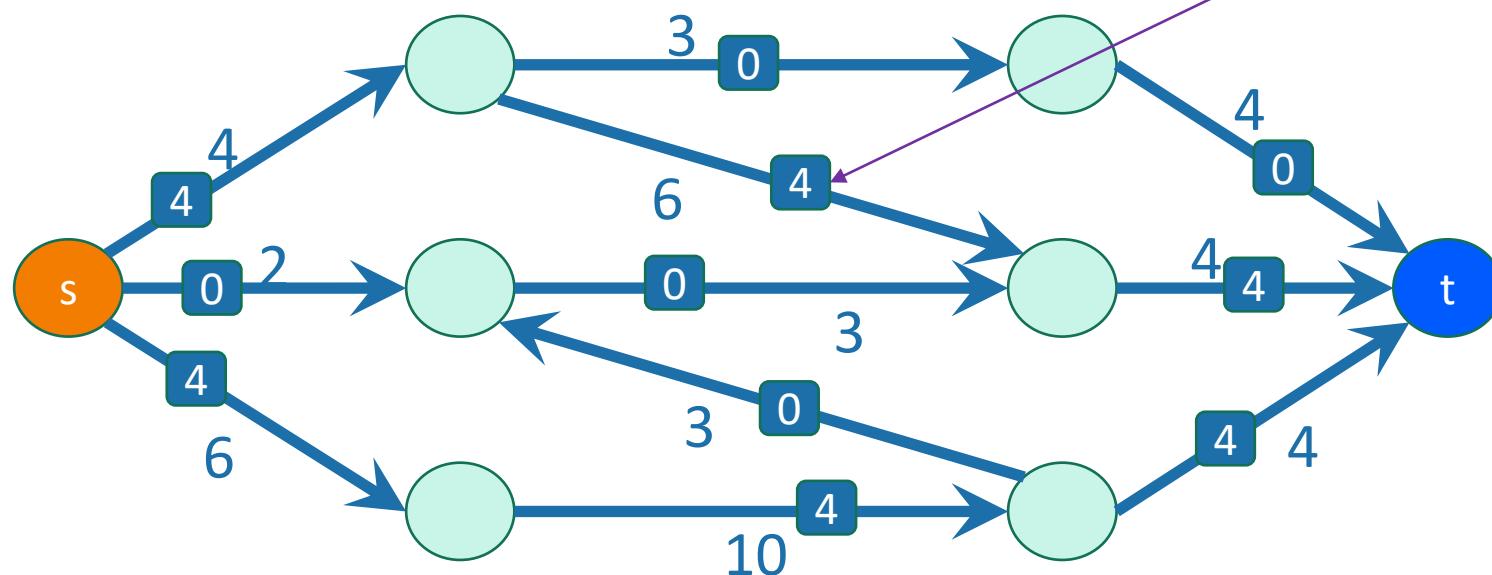
Example of Ford-Fulkerson



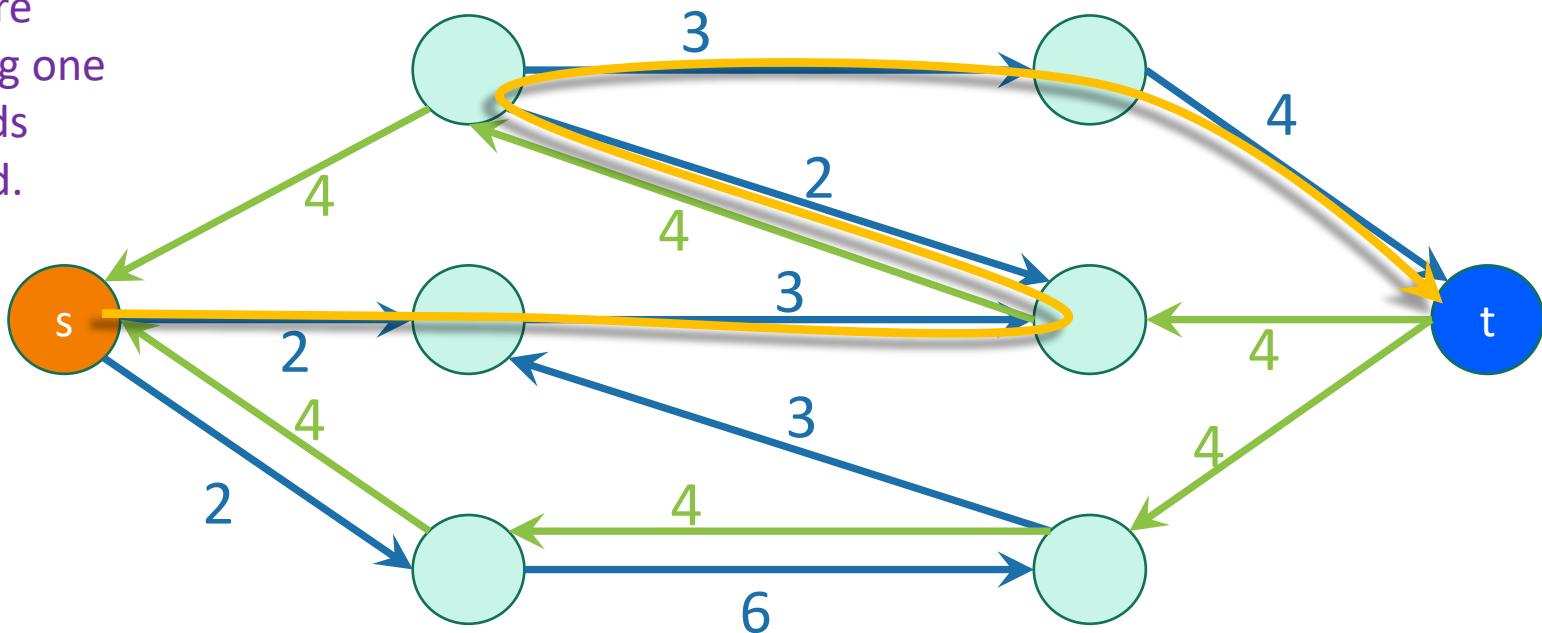
Example of Ford-Fulkerson



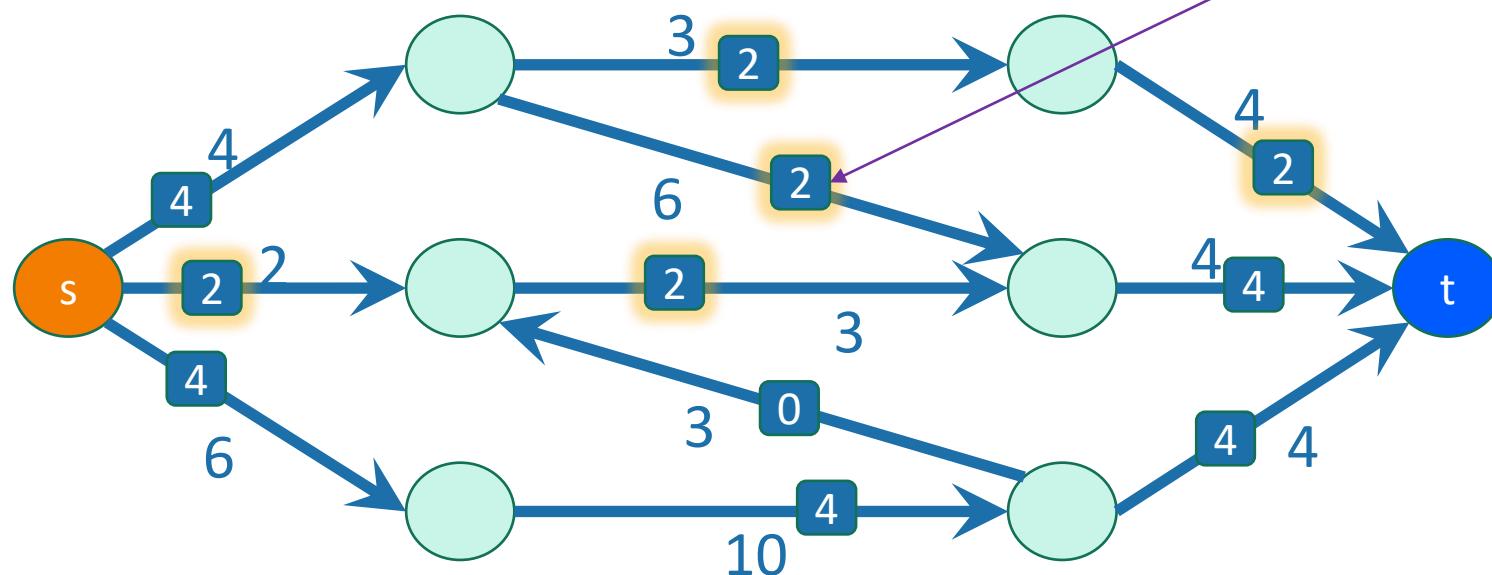
Example of Ford-Fulkerson



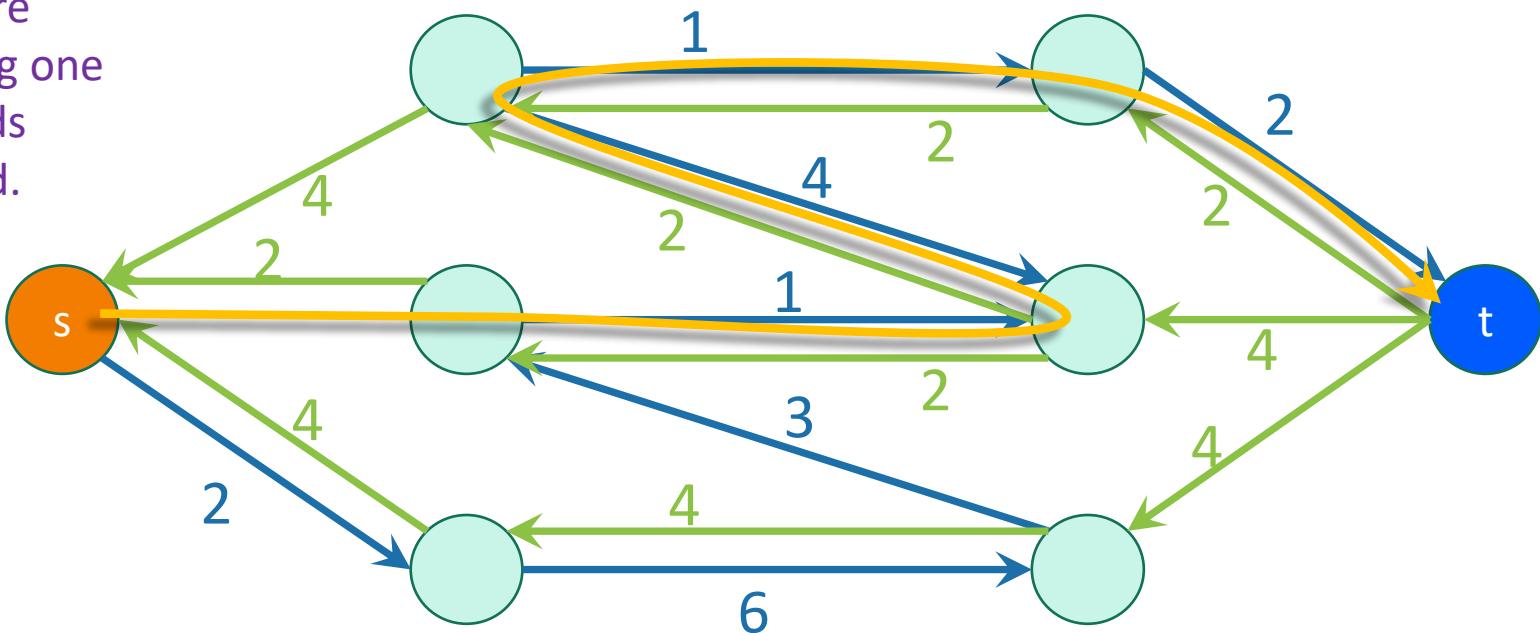
Notice that we're going back along one of the backwards edges we added.



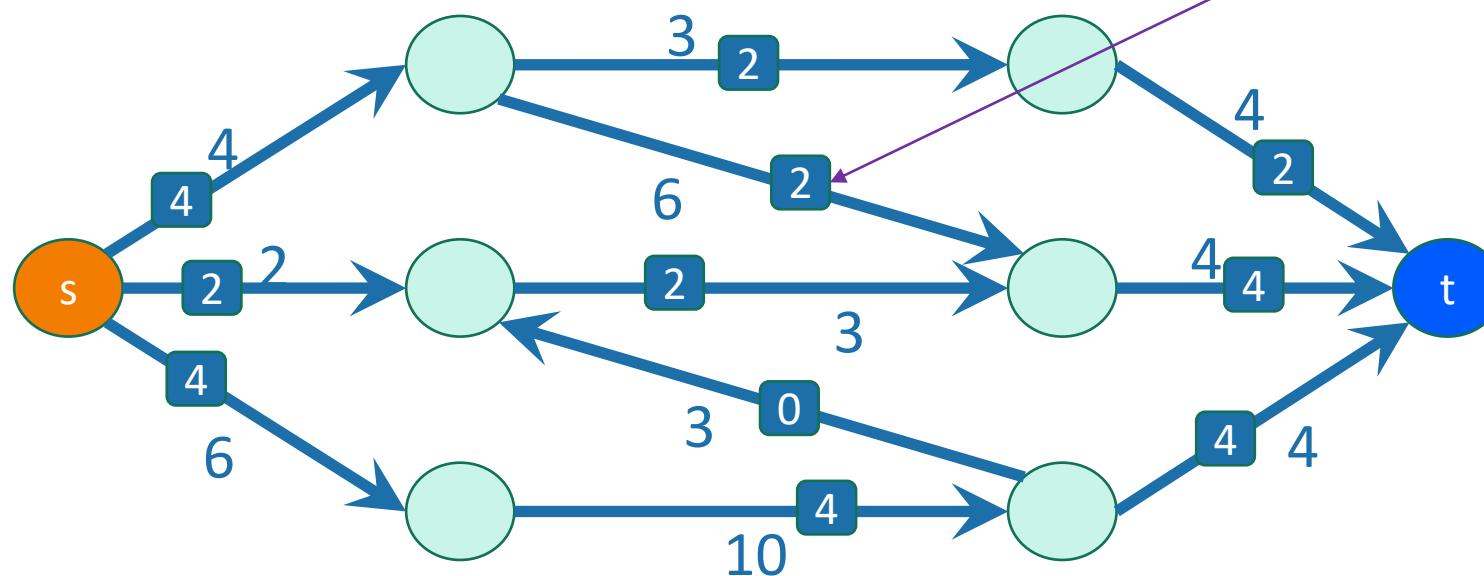
Example of Ford-Fulkerson



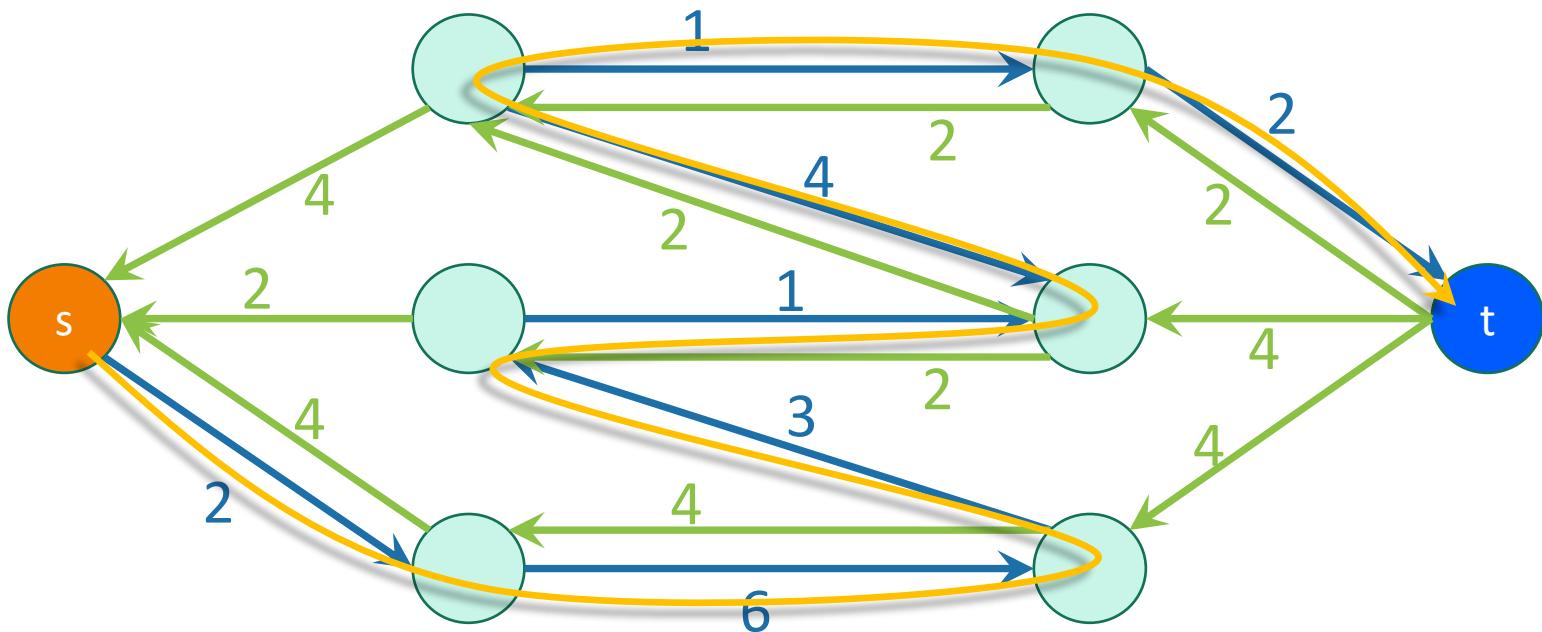
Notice that we're going back along one of the backwards edges we added.



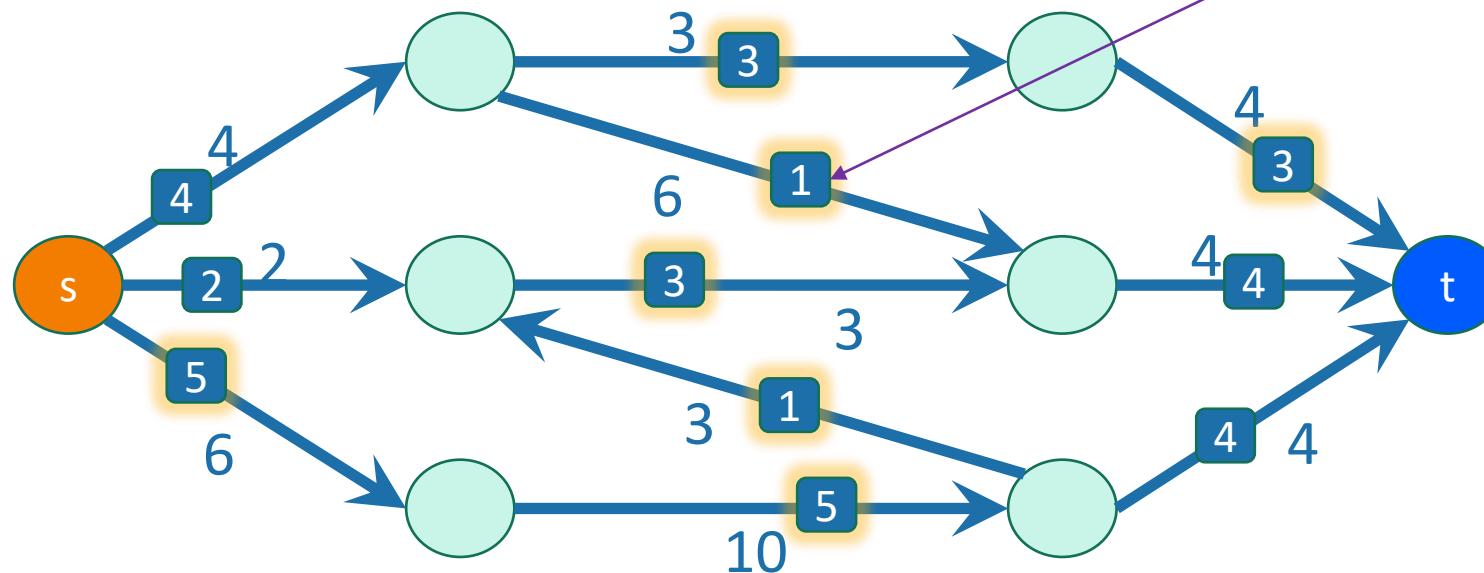
Example of Ford-Fulkerson



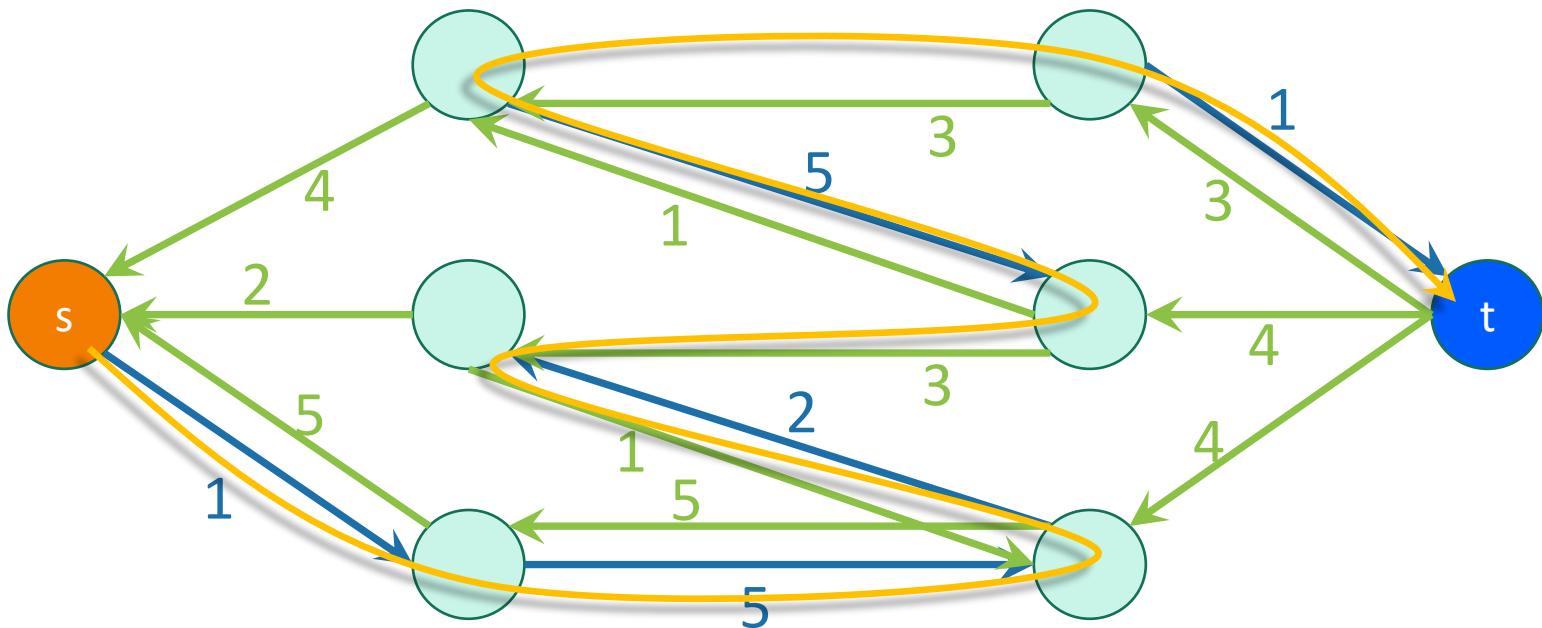
We will remove flow from this edge AGAIN.



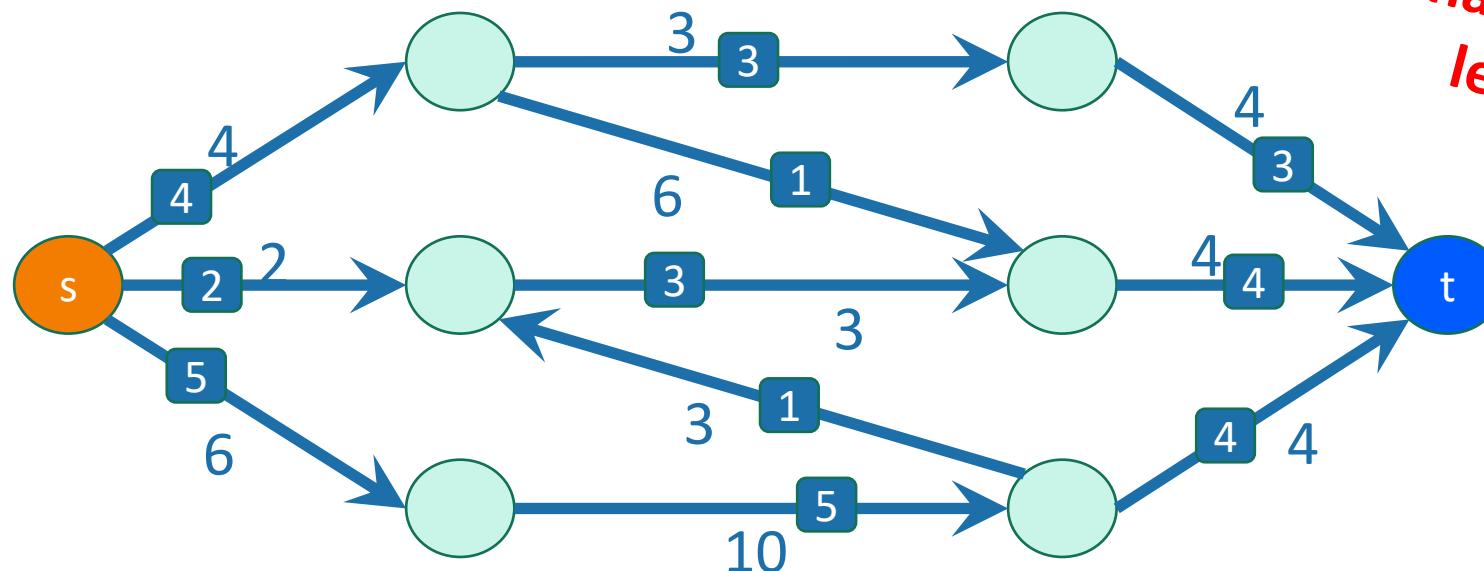
Example of Ford-Fulkerson



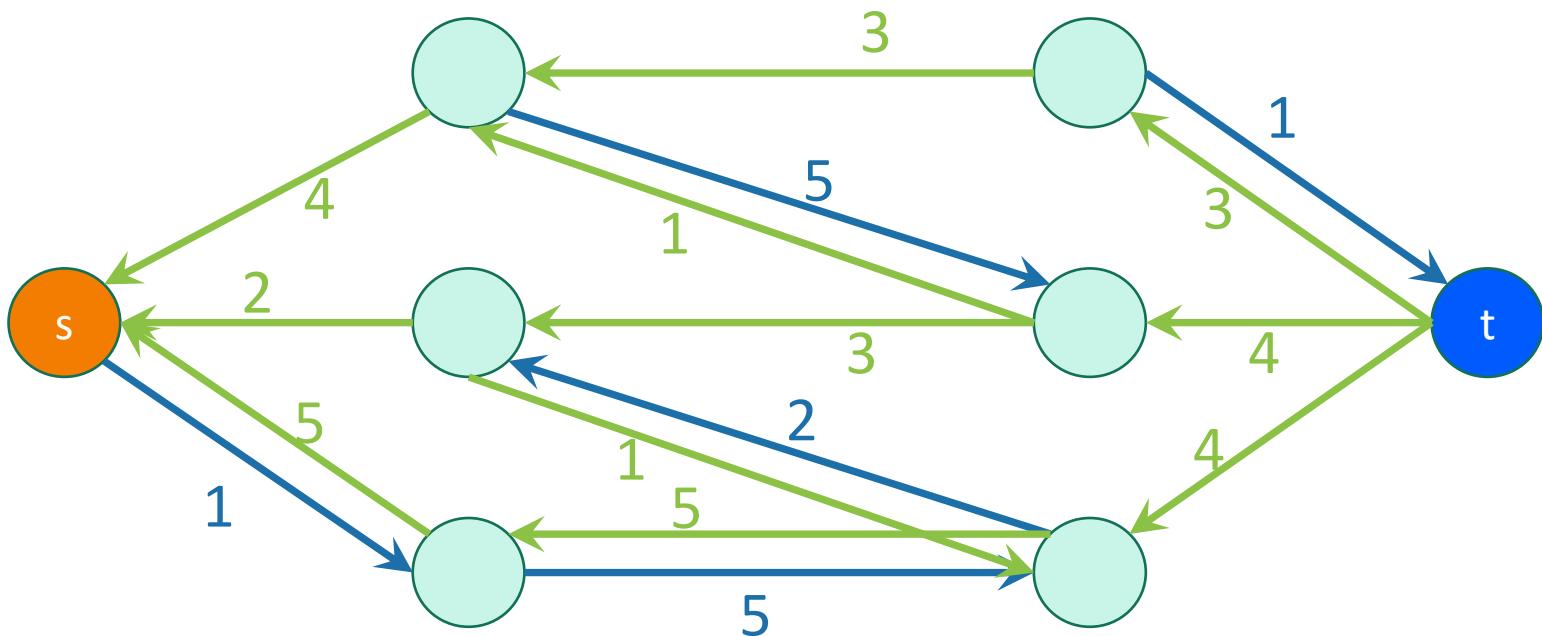
We will remove flow from this edge AGAIN.



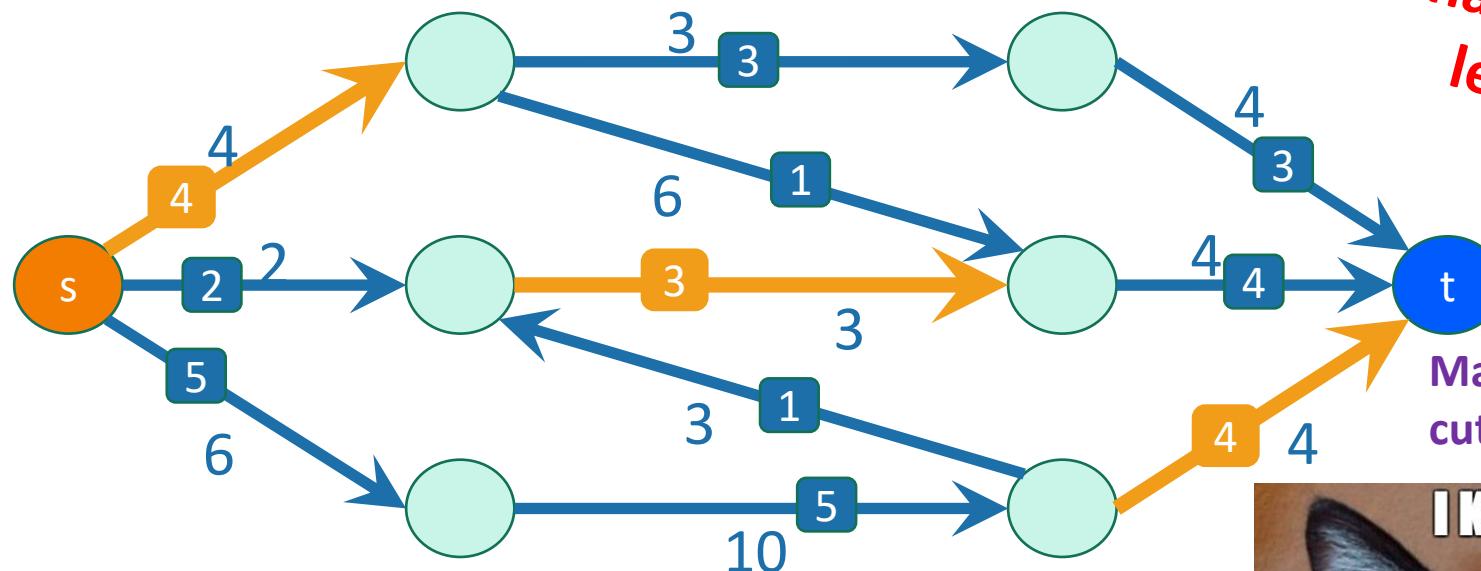
Example of Ford-Fulkerson



*Now we
have nothing
left to do!*

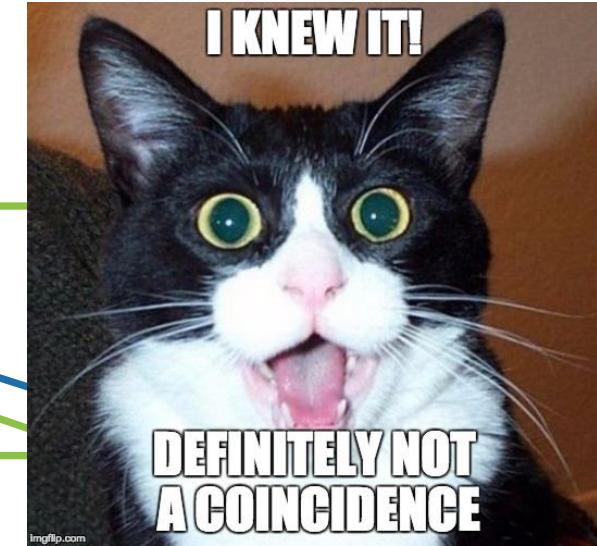


Example of Ford-Fulkerson

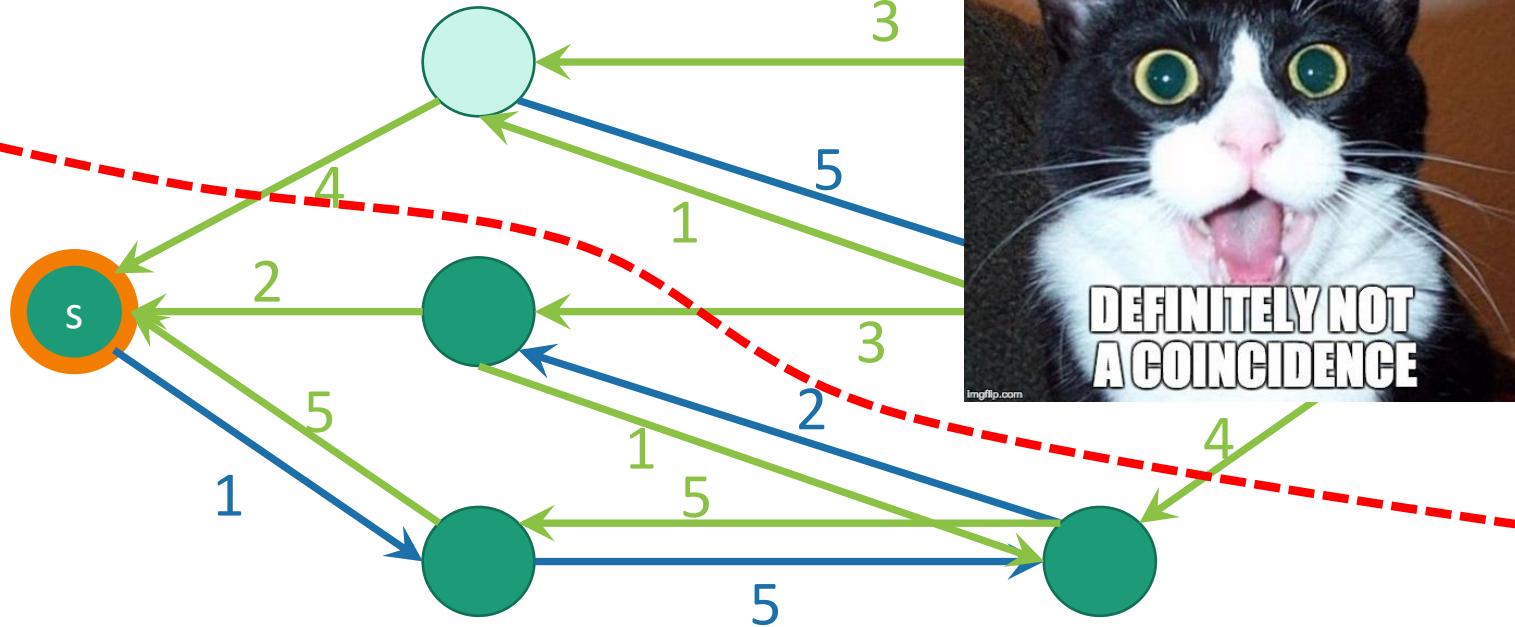


*Now we
have nothing
left to do!*

Max flow and min cut are both 11.



There's no path from s to t , and here's the cut to prove it.



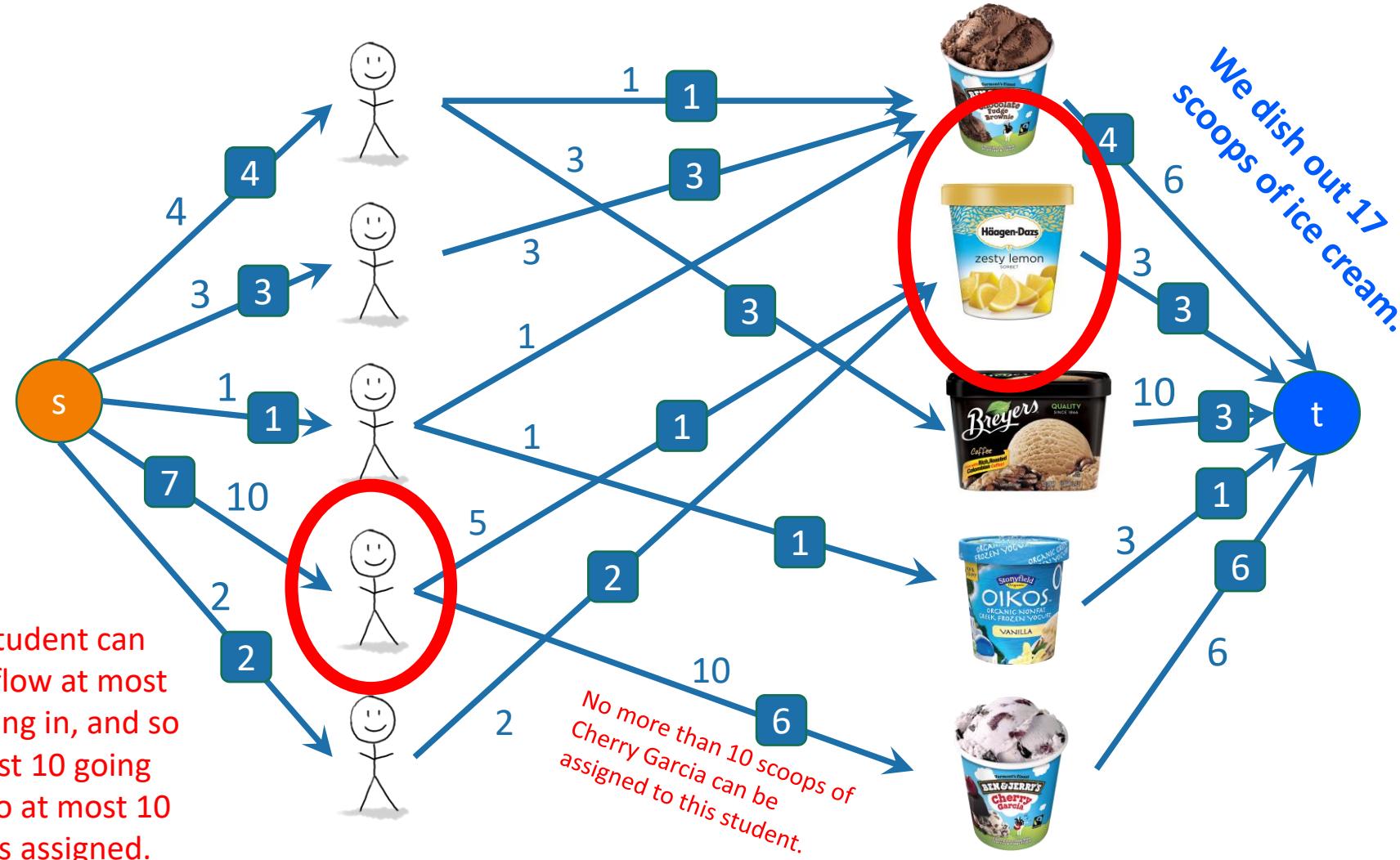
Doing Ford-Fulkerson with BFS is called
the **Edmonds-Karp algorithm**.

Theorem

- If you use BFS, the Ford-Fulkerson algorithm runs in time $O(nm^2)$.
Doesn't have anything to do with the edge weights!
- We will skip the proof in class.
- Basic idea:
 - The number of times you remove an edge from the residual graph is $O(n)$.
 - This is the hard part
 - There are at most m edges.
 - Each time we remove an edge we run BFS, which takes time $O(n+m)$.
 - Actually, $O(m)$, since we don't need to explore the whole graph, just the stuff reachable from s .

Solution via max flow

No more than 3 scoops of sorbet can be assigned.



As before, flows correspond to assignments, and max flows correspond to max assignments.

Recap

- Today we talked about s-t cuts and s-t flows.
- The **Min-Cut Max-Flow Theorem** says that minimizing the cost of cuts is the same as maximizing the value of flows.
- The Ford-Fulkerson algorithm does this!
 - Find an augmenting path
 - Increase the flow along that path
 - Repeat until you can't find any more paths and then you're done!
- An important algorithmic primitive!
 - eg, assignment problems.

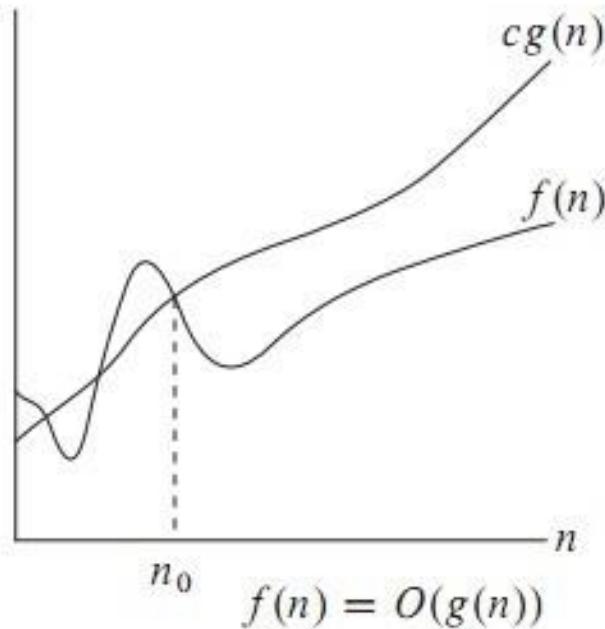
Complexity definitions

- Big-Oh
- Big-Theta
- Big - Omega

Big Oh (O)

$f(n) = O(g(n))$ *iff* there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$

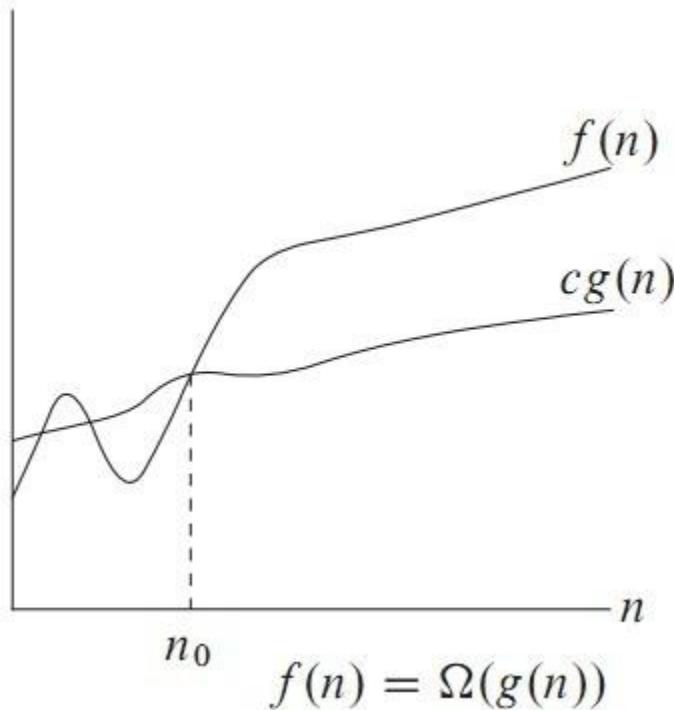
O -notation to give an **upper bound** on a function



Omega Notation

Big oh provides an asymptotic **upper** bound on a function.
Omega provides an asymptotic **lower** bound on a function.

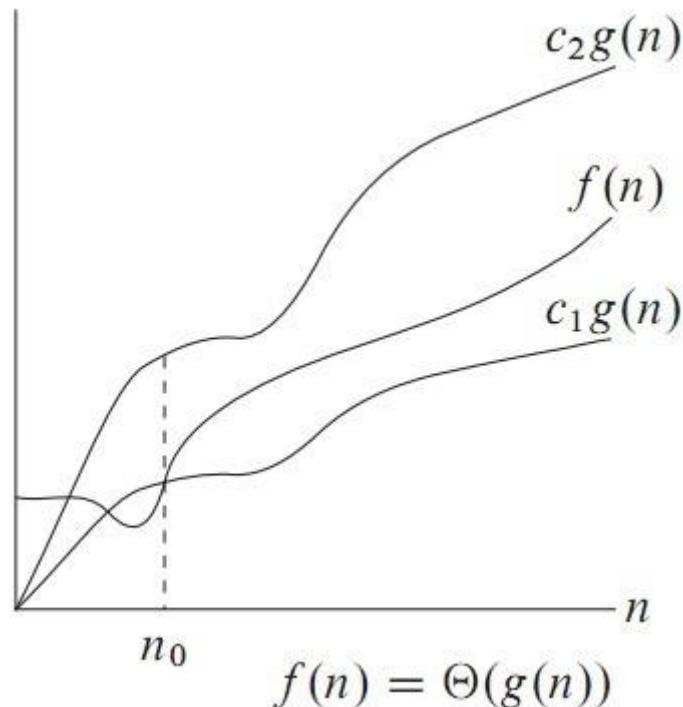
$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \} .$$



Theta Notation

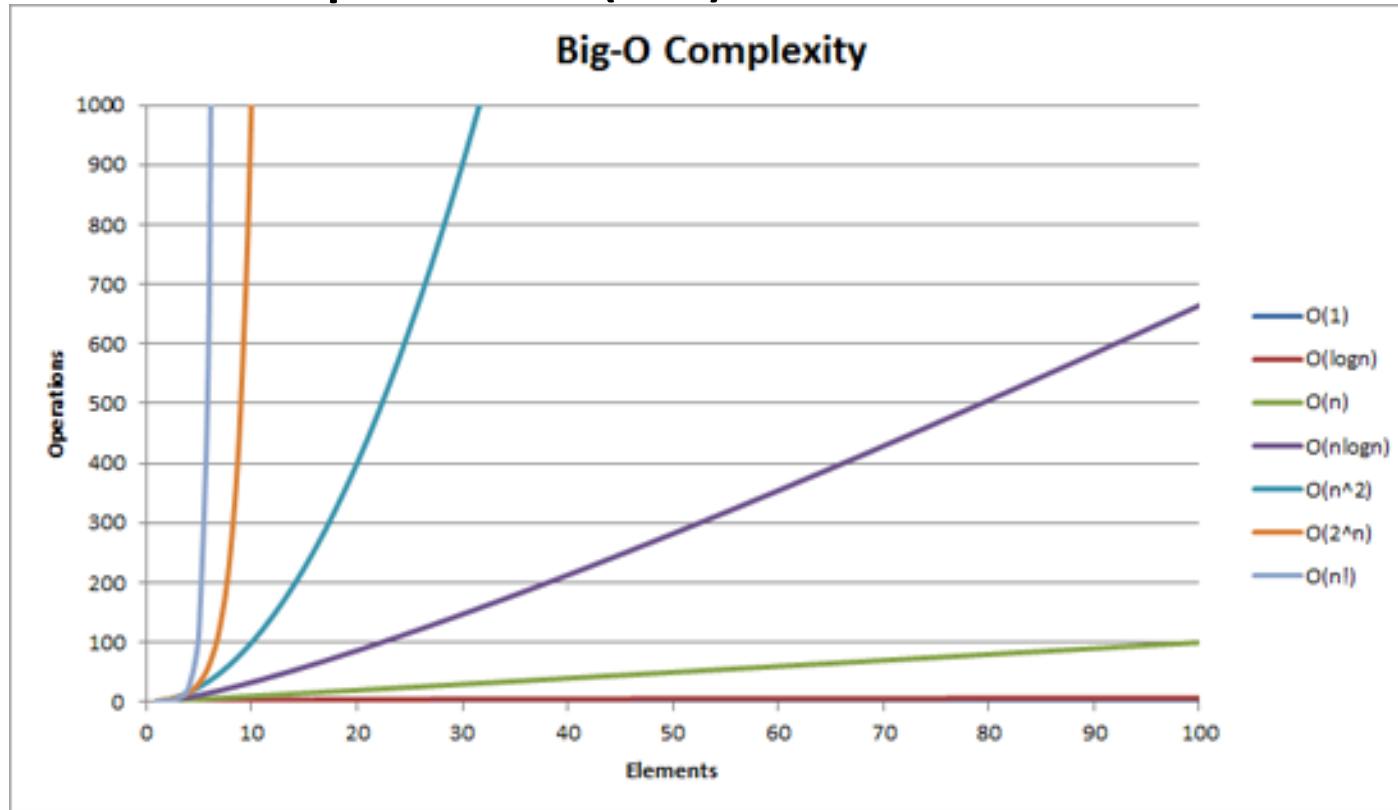
Theta notation is used when function f can be bounded **both from above and below** by the same function g

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$.¹



How bad is exponential complexity

- Fibonacci example – the recursive fib cannot even compute fib(50)



What have we learned? We've filled out a toolbox

- Tons of examples give us intuition about what algorithmic techniques might work when.
- The technical skills make sure our intuition works out.



But there's lots more out there



THANK YOU ALL!!!

Week	Date	Topic
1	21-Feb	Introduction. Some representative problems
2	28-Feb	Stable Matching
3	7-Mar	Basics of algorithm analysis.
4	14-Mar	Graphs (Project 1 announced)
5	21-Mar	Greedy algorithms-I
6	28-Mar	Greedy algorithms-II
7	4-Apr	Divide and conquer (Project 2 announced)
8	11-Apr	Dynamic Programming I
9	18-Apr	Dynamic Programming II
10	25-Apr	Network Flow-I (Project 3 announced)
11	2-May	Midterm
12	9-May	Network Flow II
13	16-May	NP and computational intractability-I
14	23-May	NP and computational intractability-II