Week 11 - Wednesday

COMP 4500

Last time

- What did we talk about last time?
- Reductions via gadgets
- Efficient certification

Questions?

Assignment 6

Logical warmup

- Ten people are marooned on a deserted island
- They gather many coconuts and put them all in a community pile
- They are so tired that they decide to divide them into ten equal piles the next morning
- One castaway wakes up hungry and decides to take his share early
- After dividing up the coconuts, he finds he is one coconut short of ten equal piles
- He notices a monkey holding one coconut
- He tries to take the monkey's coconut so that the total is evenly divisible by 10
- However, when he tries to take it, the monkey hits him on the head with it, killing him
- Later, another castaway wakes up hungry and also decides to take his share early
- On the way to the coconuts he finds the body of the first castaway and realizes that he is now be entitled to 1/9 of the total pile
- After dividing them up into nine piles he is again one coconut short of an even division and tries to take the monkey's (slightly) bloody coconut
- Again, the monkey hits the second man on the head and kills him
- Each of the remaining castaways goes through the same process, until the 10th person to wake up realizes that the entire pile for himself
- What is the smallest number of coconuts in the original pile (ignoring the monkey's)?

Three-sentence Summary of Proving Problems NP-Complete

Proving Problems are NP-Complete

NP-complete problems

- While trying to figure out if **P** = **NP**, computer scientists have considered the hardest problems in **NP**
  - What are those?
- A hardest problem ***X*** in **NP** has the following properties:
  - $X \in \mathbf{NP}$
  - For all $Y \in \mathbf{NP}$, $Y \leq_P X$
- In other words, it's a problem in **NP** that we can reduce all other problems in **NP** to
- The hardest problems in any class are its "complete" problems
- Thus, we call the hardest problems in **NP** the **NP-complete** problems

An important consequence

- **Claim:** Suppose ***X*** is an **NP-complete** problem. ***X*** is solvable in polynomial time if and only if **P** = **NP**.
- **Proof:**
  - If **P** = **NP**, then ***X*** can be solved in polynomial time, since ***X*** ∈ **NP**.
  - Conversely, suppose that ***X*** can be solved in polynomial time. For all other problems ***Y*** ∈ **NP**, $Y \leq_P X$. Thus, all problems ***Y*** can be solved in polynomial time and **NP** ⊆ **P**. Since we already know that **P** ⊆ **NP**, it would be the case that **P** = **NP**.

Why does NP-complete even exist?

- It might seem strange that there's a layer of problems that are all the "hardest" in **NP**
- Wouldn't it be possible for there to be lots of problems in **NP** that can't be reduced to each other?
- Thus, we could imagine lots of incomparable problems floating around, none of which are clearly harder than the others
- Or, there could be infinite problems in **NP**, with each one strictly harder than the previous!

Circuit

- We want a problem that builds intuition about how we might be able to encode any problem in **NP**
- Consider a circuit
  - A labeled, directed, acyclic graph with sources (no incoming edges) that are 0, 1, or the name of a variable
  - Every other node corresponds to operators $\wedge$ (AND), $\vee$ (OR), and $\sim$ (NOT)
  - A single node with no outgoing edges is the output

Circuit satisfiability

- The circuit satisfiability problem takes such a circuit as input and asks if there is an assignment of values to inputs that causes the output to be 1
  - If there is, the circuit is **satisfiable**
  - A **satisfying assignment** is one that results in this output of 1
- Circuit satisfiability is **NP-complete** because we can reduce any problem in **NP** to it

How can we reduce anything in NP to circuit satisfiability?

- Any algorithm that takes a fixed number ***n*** of bits as input and produces a "yes" or "no" answer can be represented by this kind of circuit
- The circuit is the same as an algorithm because its output is 1 on precisely the inputs for which the algorithm outputs "yes"
- If the algorithm takes a number of steps that is polynomial in ***n***, the circuit must have polynomial size
- The Cook-Levin theorem goes into careful detail about how to construct such a circuit from an algorithm

3-Satisfiability is NP-complete

- **Proof:**
  - 3-SAT is in **NP** since we can verify in polynomial time that a truth assignment satisfies a given set of clauses.
  - By reducing circuit satisfiability to 3-SAT, we will thus prove that 3-SAT is **NP-complete**.
  1. Turn any circuit into an equivalent instance of SAT with **at most** 3 variables per clause
  2. Turn any instance of SAT where each clause has at most 3 variables into an equivalent instance with **exactly** 3 variables

Step 1: Convert circuit *K* to SAT

- Make variable x_v for each node ***v*** of ***K*** to hold the truth value for that node
  - If ***v*** is a NOT and its entering edge comes from ***u***, then we need $x_v = \overline{x_u}$, for that, we add clauses $(x_v \vee x_u)$ and $(\overline{x_v} \vee \overline{x_u})$
  - If ***v*** is an OR and its entering edges come from ***u*** and ***w***, we need $x_v = x_u \vee x_w$, for that we add clauses $(x_v \vee \overline{x_u})$, $(x_v \vee \overline{x_w})$, and $(\overline{x_v} \vee x_u \vee x_w)$
  - If ***v*** is an AND and its entering edges come from ***u*** and ***w***, we need $x_v = x_u \wedge x_w$, for that we add clauses $(\overline{x_v} \vee x_u)$, $(\overline{x_v} \vee x_w)$, and $(x_v \vee \overline{x_u} \vee \overline{x_w})$
  - If ***v*** is a 1 or a 0, we set it to the clause x_v or $\overline{x_v}$, respectively
  - If ***v*** is the output node, we add the clause x_v to force it to 1

Step 2: Converting the SAT to 3-SAT

- 3-SAT has exactly three variables for each clause, but some of our clauses have one or two variables
- Create four new variables: z_1, z_2, z_3, z_4
- Create four clauses for each: $(\overline{z_1} \vee z_3 \vee z_4)$, $(\overline{z_1} \vee \overline{z_3} \vee z_4)$, $(\overline{z_1} \vee z_3 \vee \overline{z_4})$, and $(\overline{z_1} \vee \overline{z_3} \vee \overline{z_4})$
- These clauses force $z_1 = z_2 = 0$
- Then, for two-term clause, we OR z_1 with it, and for any single term clause we OR $z_1 \vee z_2$ with it
- This new formula is satisfiable if and only if the original circuit was, and we were able to construct it in polynomial time.
- Thus, circuit SAT $\leq_P$ 3-SAT, and 3-SAT is **NP-complete**.

■

We get a bunch for free!

- Earlier, we showed:
  - 3-SAT $\leq_P$ independent set $\leq_P$ vertex cover $\leq_P$ set cover
- By the transitivity of polynomial-time reduction, circuit SAT is reducible to all of these problems
- Thus, all of these problems are **NP-complete**

Strategy for proving problems NP-complete

- Given a problem ***X*** that might be **NP-complete**

1. Prove that ***X*** ∈ **NP**
2. Choose a problem ***Y*** that is known to be **NP-complete**
3. Prove that ***Y*** $\leq_P$ ***X***
   - Specifically, consider an arbitrary instance s_Y of ***Y*** and show how to construct in polynomial time an instance s_X of ***X*** such that:
     a. If s_Y is a "yes" instance of ***Y***, then s_X is a "yes" instance of ***X***
     b. If s_X is a "yes" instance of ***X***, then s_Y is a "yes" instance of ***Y***

Review

Dynamic Programming

Weighted interval scheduling

- The **weighted interval scheduling** problem extends interval scheduling by attaching a weight (usually a real number) to each request
- Now the goal is not to maximize the **number** of requests served but the total **weight**
- Our greedy approach is worthless, since some high value requests might be tossed out
- We could try all possible subsets of requests, but there are **exponential** of those
- **Dynamic programming** will allow us to save parts of optimal answers and combine them efficiently

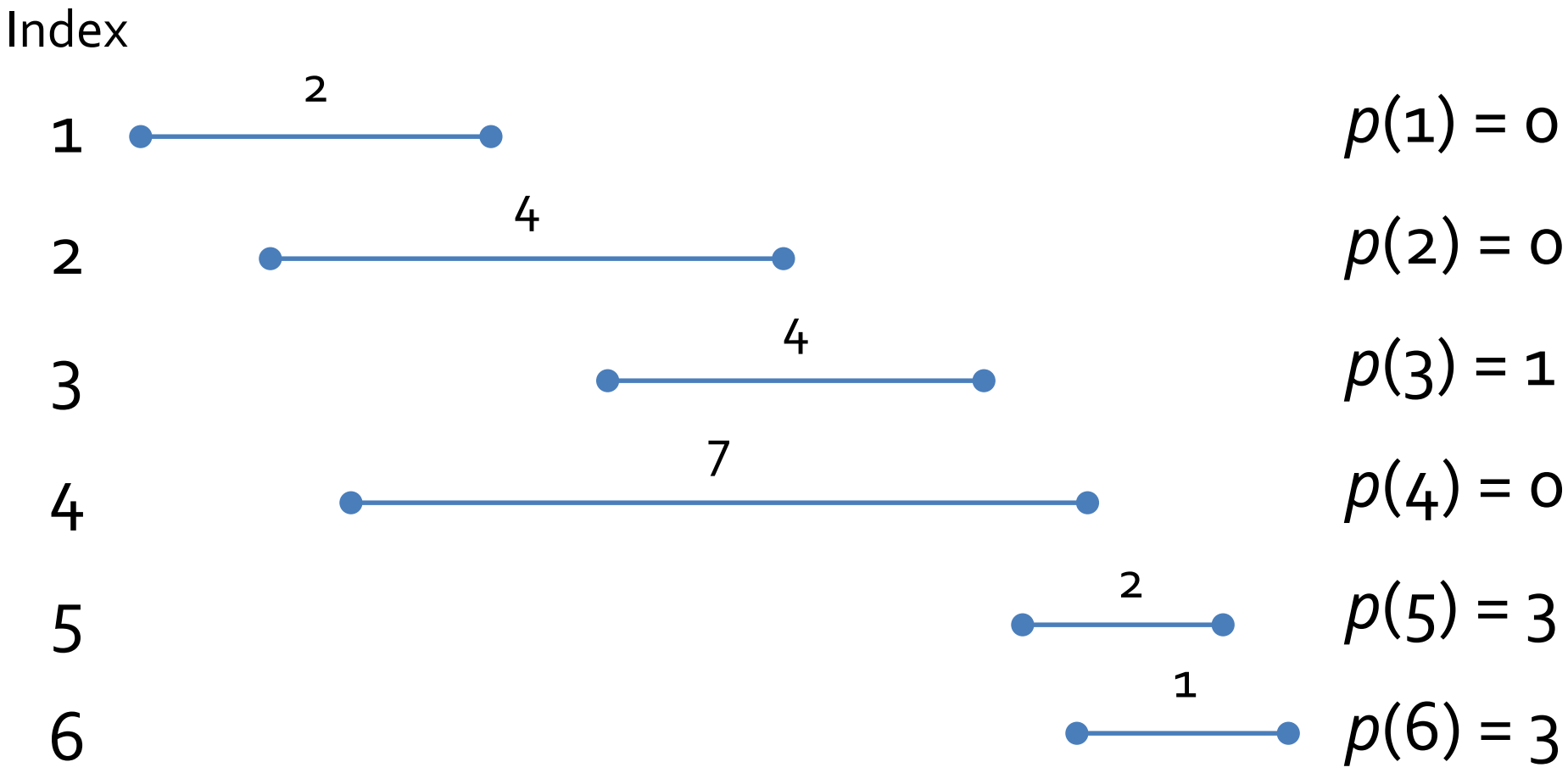
Notation

- We have n requests labeled 1, 2,…, n
- Request i has a start time s_i and a finish time f_i
- Request i has a value v_i
- Two intervals are **compatible** if they don't overlap

Designing the algorithm

- Let's go back to our intuition from the unweighted problem
- Imagine that the requests are sorted by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$
- We say that request i comes before request j if $i < j$, giving a natural left-to-right order
- For any request j, let $p(j)$ be the largest index $i < j$ such that request i ends before j begins
  - If there is no such request, then $p(j) = 0$

$p(j)$ examples



Iterative solution to find value of weighted interval scheduling

- Iterative-Compute-Opt
  - $M[0] = 0$
  - For $j = 1$ up to n
    - $M[j] = \max(v_j + M[p(j)], M[j-1])$

- Algorithm is $O(n)$

Algorithm for solution

- Find-Solution(j, M)
  - If $j = 0$ then
    - Output nothing
  - Else if $v_j + M[p(j)] \geq M[j-1]$ then
    - Output j together with the result of Find-Solution($p(j)$)
  - Else
    - Output the result of Find-Solution($j-1$)
- Algorithm is $O(n)$

Why is this dynamic programming?

- The key element that separates dynamic programming from divide-and-conquer is that you have to *keep* the answers to subproblems around
- It's not simply a one-and-done situation
- Based on which intervals overlap with which other intervals, it's hard to predict when you'll need an earlier $M[j]$ value
- Thus, dynamic programming can often give us polynomial algorithms *but* with linear (and sometimes even larger) space requirements

Informal guidelines

- Weighted interval scheduling follows a set of informal guidelines that are essentially universal in dynamic programming solutions:
  1. There are only a polynomial number of subproblems
  2. The solution to the original problem can easily be computed from (or is one of) the solutions to the subproblems
  3. There is a natural ordering of subproblems from "smallest" to "largest"
  4. There is an easy-to-compute recurrence that lets us compute the solution to a subproblem from the solutions of smaller subproblems

Subset sum

- Let's say that we have a series of ***n*** jobs that we can run on a single machine
- Each job ***i*** takes time w_i
- We must finish all jobs before time ***W***
- We want to keep the machine as busy as possible, working on jobs until as close to ***W*** as we can

A new recurrence

- If job n is not in the optimal set, OPT(n, W) = OPT($n - 1$, W)
- If job n is in the optimal set, OPT(n, W) = w_n + OPT($n - 1$, $W - w_n$)
- We can make the full recurrence for all possible weight values:
  - If $w < w_i$, then OPT(i, w) = OPT($i - 1$, w)
  - Otherwise, OPT(i, w) = max(OPT($i - 1$, w), w_i + OPT($i - 1$, $w - w_i$))

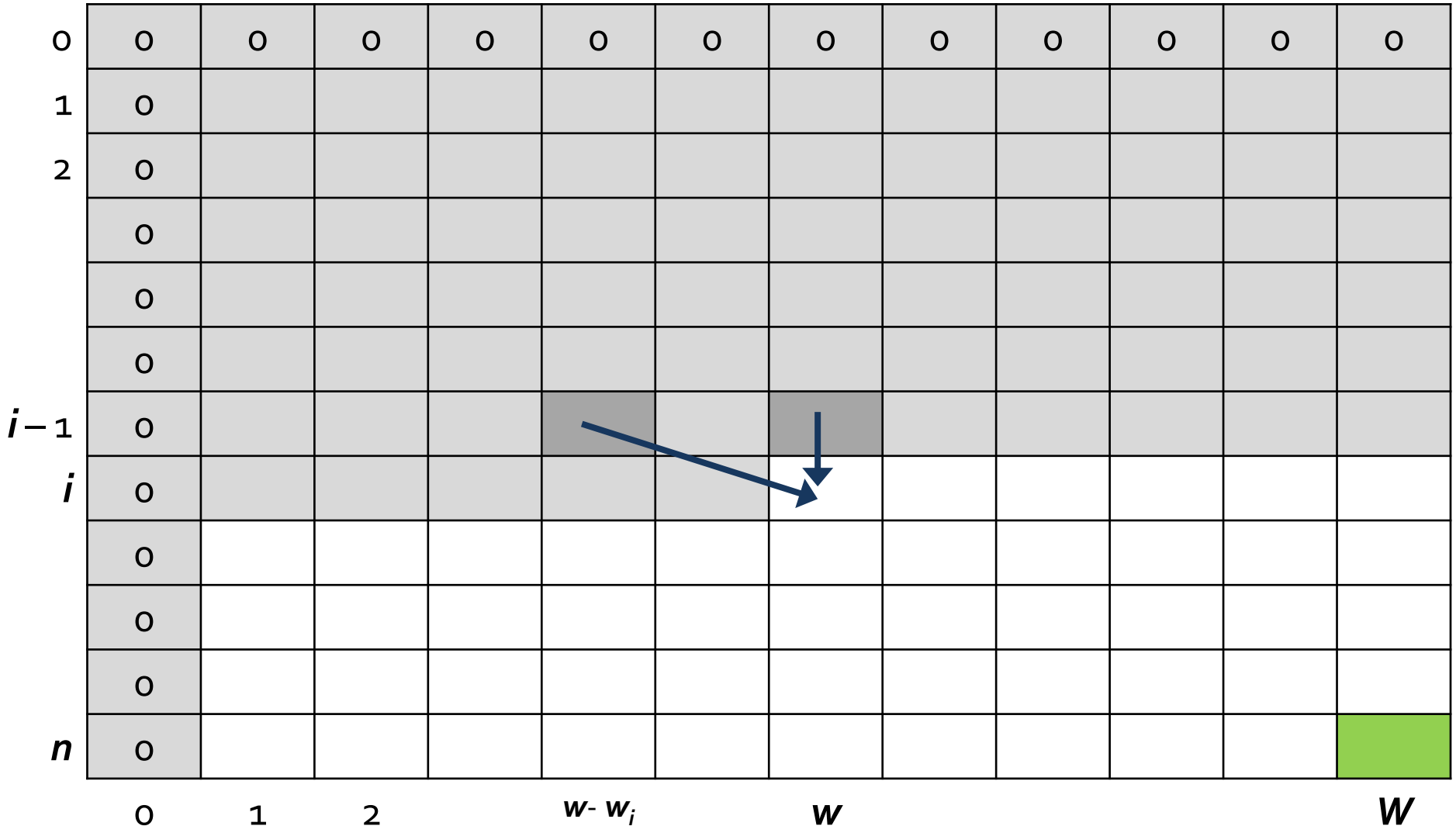
Subset-Sum(n, W)

- Create 2D array $M[0 \ldots n][0 \ldots W]$
- For w from 1 to W
  - Initialize $M[0][w] = 0$
- For i from 1 to n
  - For w from 0 to W
    - If $w < w_i$, then
      - $OPT(i, w) = OPT(i - 1, w)$
    - Else
      - $OPT(i, w) = \max(OPT(i - 1, w), w_i + OPT(i - 1, w - w_i))$
- Return $M[n][W]$

What does that look like?

- We're building a big 2D array
- Its size is ***nW***
  - ***n*** is the number of items
  - ***W*** is the maximum weight
  - Actually, it's got one more row and one more column, just to make things easier
- The book makes this array with row 0 at the bottom
- I've never seen anyone else do that
- I'm going to put row 0 at the top

Table *M* of OPT values



Running time

- The algorithm has a simple nested loop
  - The outer loop runs $n + 1$ times
  - The inner loop runs $W + 1$ times
- The total running time is $O(nW)$
- The space needed is also $O(nW)$
- Note that this time is not polynomial in terms of n
- It's polynomial in n and W, but W is the maximum weight
  - Which could be huge!
- We call running times like this **pseudo-polynomial**
- Things are fine if W is similar to n, but it could be huge!

Subset sum example

- Weights: 1, 4, 8, 2, 10
- Maximum: 15
- Create the table to find all of the optimal values that include items 1, 2,…, ***i*** for every possible weight ***w*** up to 15

Table to fill in

i	w_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1																
2	8																
3	4																
4	2																
5	10																

Knapsack

- The knapsack problem is a classic problem that extends subset sum a little
- As before, there is a maximum capacity W and each item has a weight w_i
- Each item also has a value v_i
- The goal is to maximize the value of objects collected without exceeding the capacity
- …like Indiana Jones trying to put the most valuable objects from a tomb into his limited-capacity knapsack

An easy extension

- The knapsack problem is really the same problem, except that we are concerned with maximum **value** instead of maximum **weight**
- We need only to update the recurrence to keep the maximum value:
  - If $w < w_i$, then OPT(i, w) = OPT($i - 1$, w)
  - Otherwise, OPT(i, w) = max(OPT($i - 1$, w), v_i + OPT($i - 1$, $w - w_i$))

Knapsack example

- Items (w_i, v_i):
  - (1, 15)
  - (5, 10)
  - (3, 9)
  - (4, 5)
- Maximum weight: 8
- Create the table to find all of the optimal values that include items 1, 2,…, ***i*** for every possible weight ***w*** up to 8

Fill in the table

i	w_i	v_i	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0	0	0
1	1	15	0								
2	5	10	0								
3	3	9	0								
4	4	5	0								

Alignment

- An alignment is a list of matches between characters in strings ***X*** and ***Y*** that doesn't cross
- Consider:
  - `stop-`
  - `-tops`
- This alignment is (2,1), (3,2), (4,3)

Alignment cost

- Some optimal alignment will have the lowest cost
- Cost:
  - Gap penalty $\delta > 0$, for every gap
  - Mismatch cost α_{pq} for aligning p with q
    - α_{pp} is presumably 0 but does not have to be
  - Total cost is the sum of the gap penalties and mismatch costs

Formulating the recurrence

- Let OPT(***i***, ***j***) be the minimum cost of an alignment of the first ***i*** characters in ***X*** to the first ***j*** characters in ***Y***
- In case 1, we would have to pay a matching cost of matching the character at ***i*** to ***j***
- In cases 2 and 3, you will pay a gap penalty

$$\mathrm{OPT}(i,j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1) \\ \delta + \mathrm{OPT}(i-1, j) \\ \delta + \mathrm{OPT}(i, j-1) \end{cases}$$

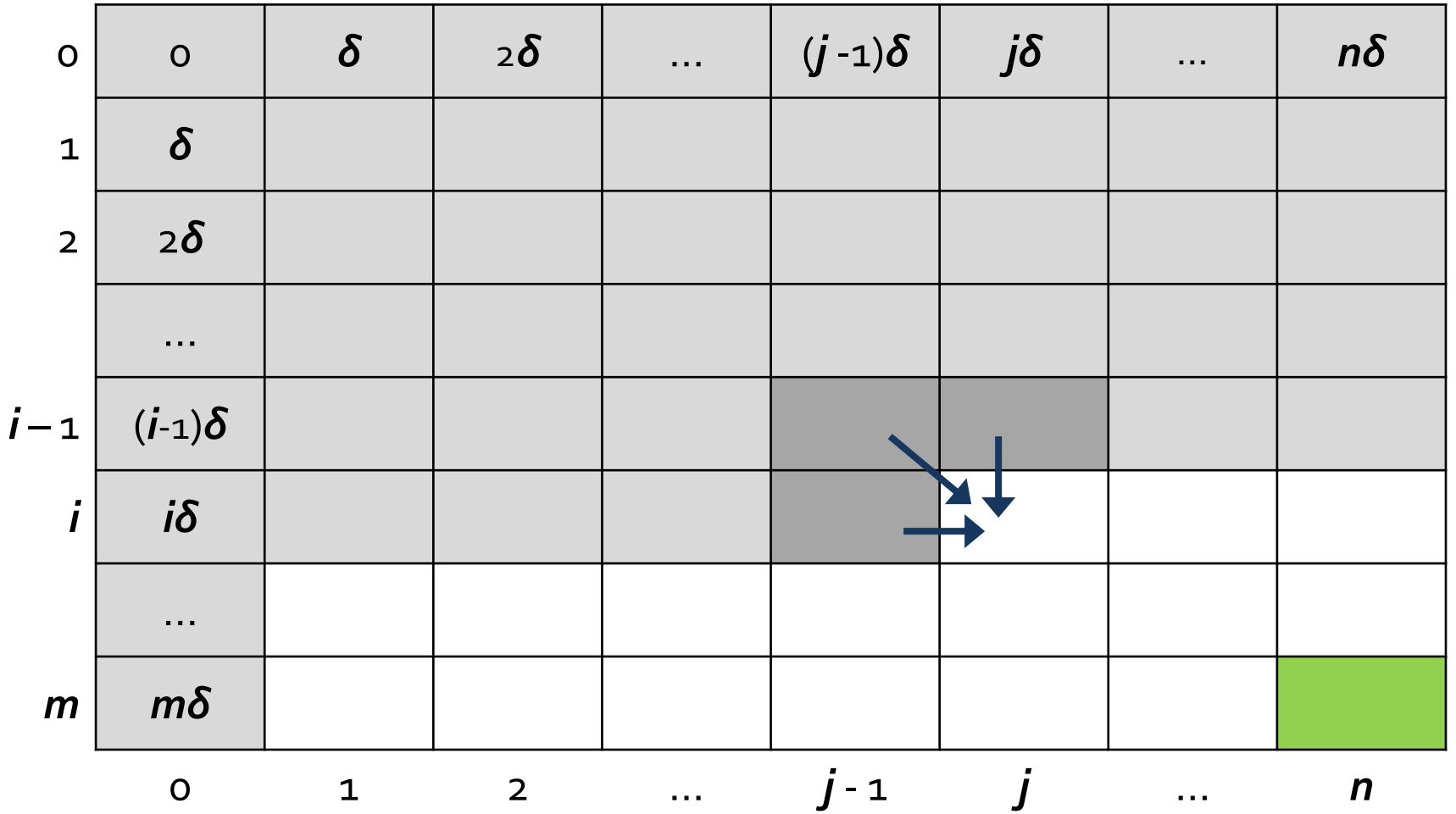
Now what?

- We do our usual thing
- Build up a table of values with $m + 1$ rows and $n + 1$ columns
- In row 0, column i has value $i\delta$ to build up strings from the empty string
- In column 0, row i has value $i\delta$ to build up strings from the empty string
- The other entries (i,j) can be computed from $(i - 1, j - 1)$, $(i - 1, j)$, $(i, j - 1)$

Alignment(*X*, *Y*)

- Create array ***A***[0…***m***][0…***n***]
- For ***i*** from 0 to ***m***
  - Set ***A***[***i***][0]= ***iδ***
- For ***j*** from 0 to ***n***
  - Set ***A***[0][***j***]= ***jδ***
- For ***i*** from 1 to ***m***
  - For ***j*** from 1 to ***n***
    - Set ***A***[***i***][***j***]= min($\alpha_{x_i y_j}$+***A***[***i***-1][***j***-1], ***δ*** + ***A***[***i***-1][***j***], ***δ*** + ***A***[***i***][***j***- 1])
- Return ***A***[***m***][***n***]

Table *A* of OPT values



Sequence alignment example

- Find the minimum cost to align:
  - "garbage"
  - "ravaged"
- The cost of an insertion (or deletion) δ is 1
- The cost of replacing any letter with another letter is 1
- The cost of "replacing" any letter with itself is 0

Fill in the table

		g	a	r	b	a	g	e
	0	1	2	3	4	5	6	7
r	1							
a	2							
v	3							
a	4							
g	5							
e	6							
d	7							

Maximum Flow

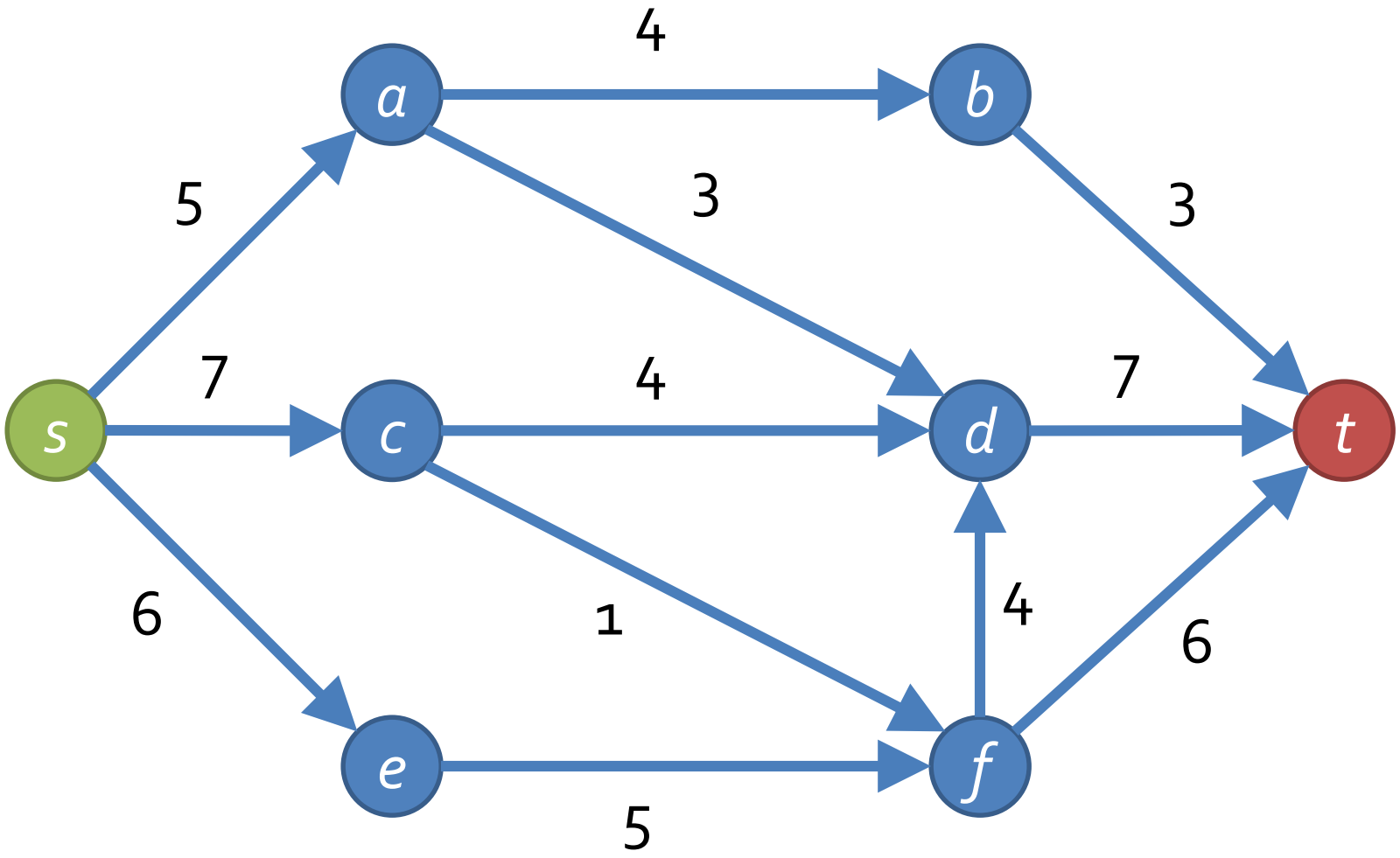
Flow networks

- A **flow network** is a weighted, directed graph with positive edge weights
  - Think of the weights as **capacities**, representing the maximum units that can flow across an edge
  - It has a **source *s*** (where everything comes from)
  - And a **sink *t*** (where everything goes to)
- Some books refer to this kind of flow network specifically as an ***st***-flow network

Maximum flow

- A common flow problem is to find the **maximum flow**
- A maximum flow is a flow such that the amount leaving s and the amount going into t is as large as possible
- In other words:
  - The maximum amount of flow gets from ***s*** to ***t***
  - No edge has more flow than its capacity
  - The flow going into every node (except ***s*** and ***t***) is equal to the flow going out

Flow network



Ford-Fulkerson algorithm

- Ford-Fulkerson is a family of algorithms for finding the maximum flow

1. Start with zero flow on all edges
2. Find an augmenting path (increasing flow on forward edges and decreasing flow on backwards edges)
3. If you can still find an augmenting path in the residual graph, go back to Step 2

Bipartite Matching

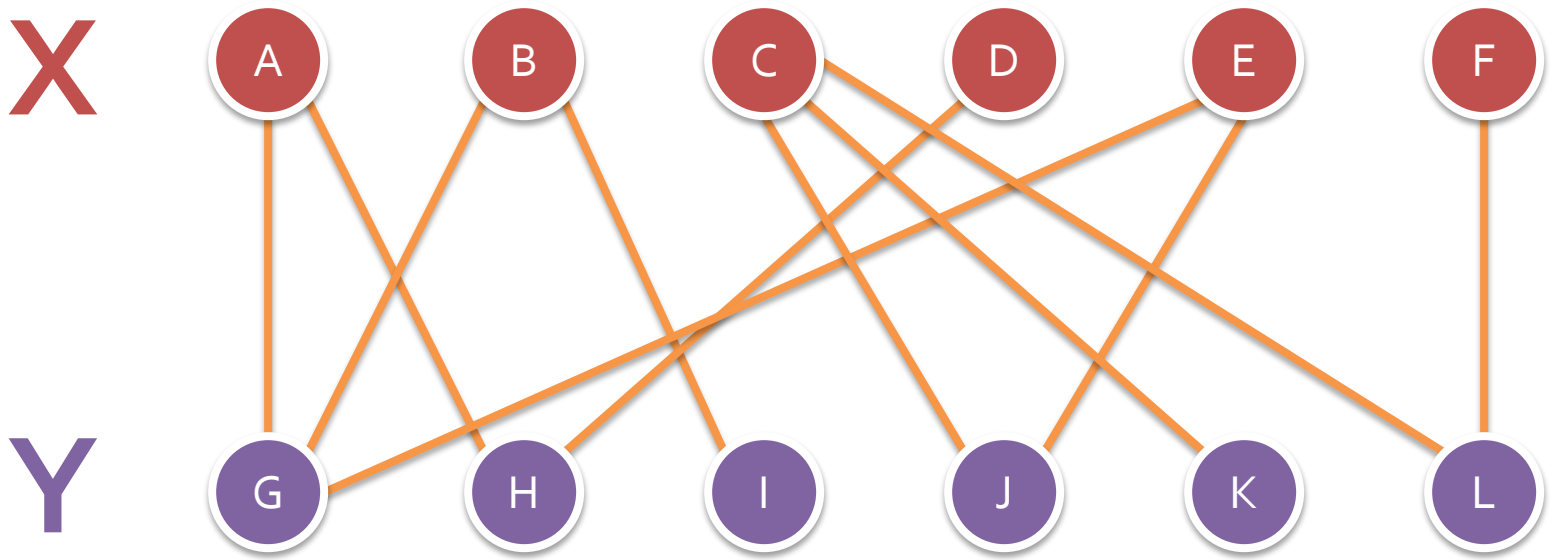
Bipartite graphs

- Recall that a bipartite graph is one whose nodes can be divided into two disjoint sets X and Y
- Every edge has one end in set X and the other in set Y
  - There are no edges from a node inside set X to another node in set X
  - There are no edges from a node inside set Y to another in set Y
- Equivalently, a graph is bipartite if and only if it contains no odd cycles

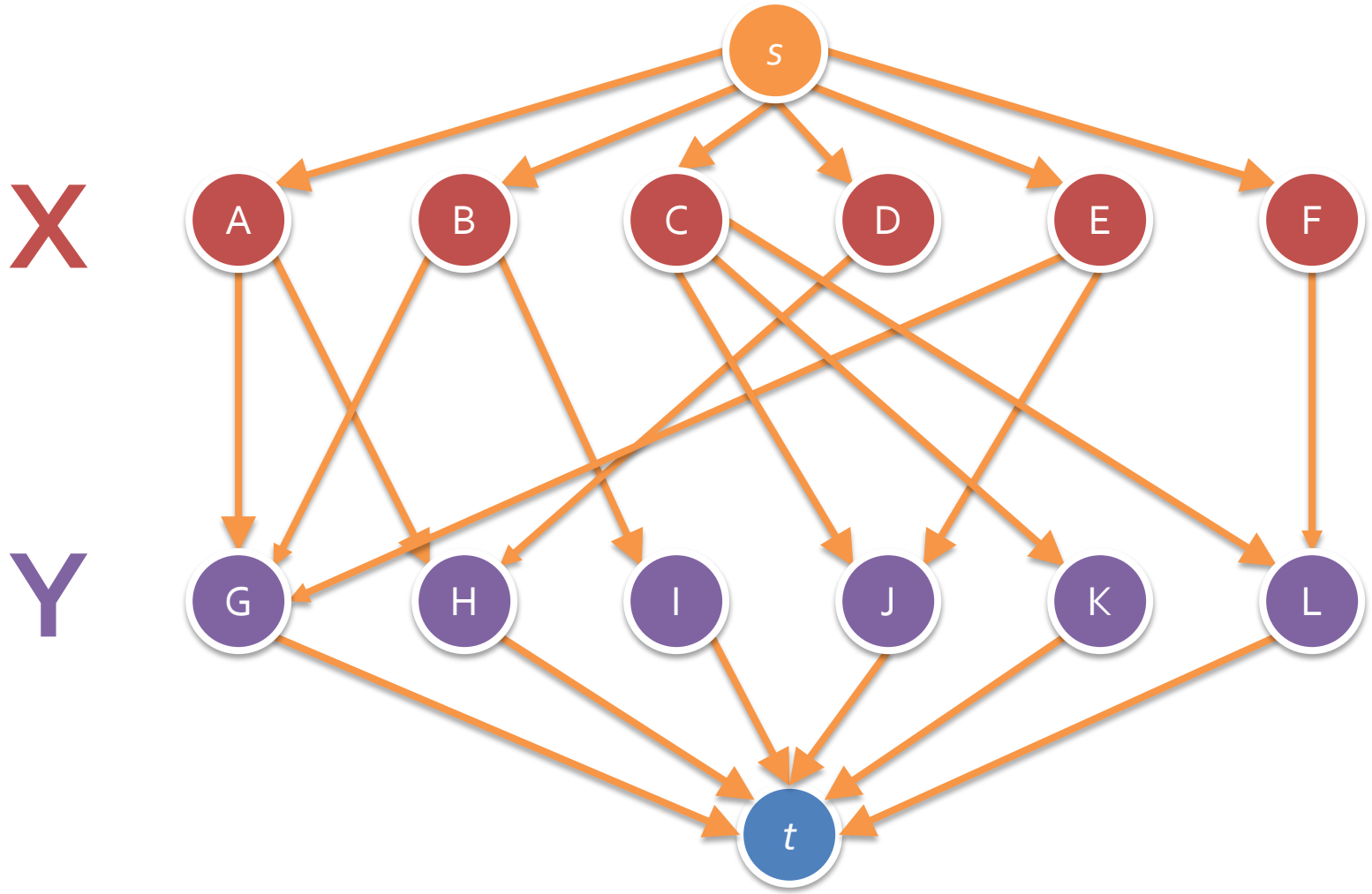
Maximum matching

- **Matching** means pairing up nodes in set X with nodes in set Y
- A node can only be in one pair
- A **perfect matching** is when every node in set X and every node in set Y is matched
- It is not always possible to have a perfect matching
- We can still try to find a **maximum matching** in which as many nodes are matched up as possible

Bipartite matching problem



Maximum flow problem



An easy change

- Take a bipartite graph *G* and turn it into a directed graph *G'*
- Create a source node ***s*** and a sink node ***t***
- Connect directed edges from the source to all the nodes in set X
- Connect directed edges from all the nodes in set Y to the sink
- Change all the undirected edges from X to Y to directed edges from X to Y
- Set the capacities of all edges to 1

Algorithmic changes

- We run the Ford-Fulkerson algorithm to find the maximum flow on our new graph
- Since all edges from X to Y have capacity 1, they will either have a flow of 1 or of 0
- If they have a flow of 1, they are in the matching
- If they have a flow of 0, they aren't
- The maximum flow value tells us how many nodes are matched

Maximal matching

- To make the algorithm go faster, we can start with a **maximal matching**
- A maximal matching is not necessarily maximum, but you can't add edges to it directly without removing other edges
- In essence, arbitrarily match unmatched nodes until you can't anymore
- Then start the process of looking for augmenting paths

Matching algorithm

1. Come up with a legal, maximal matching
2. Take an **augmenting path** that starts at an unmatched node in X and ends at an unmatched node in Y
3. If there is such a path, switch all the edges along the path from being in the matching to being out and vice versa
4. If there is another augmenting path, go back to Step 2

Quiz

Upcoming

Next time…

- Exam 3
- After that:
  - Sequencing problems
  - Partitioning problems
  - Graph coloring
  - Numerical problems
  - Co-NP

Reminders

- **No class Friday!**
- Work on Assignment 6
- **Study for Exam 3**
  - **In class on Monday**
- **For next Wednesday**, read 8.5, 8.7, 8.8, and 8.9
  - Your three-sentence summary should list **all** of the different NP-complete problems