

CPSC 320

July 3rd - Introduction & The Stable Marriage Problem (SMP)

Introduction

- Instructor: Susanne Bradley
- Problem solving
 - General algorithm design techniques, how to use them, how to identify them
 - Recognizing similarities
 - Identifying promising data structures and computational techniques
 - Reasoning about the correctness and efficiency
- Lectures largely involve problem-solving (solving problems independently & w/ peers, TA and instructor may help)
 - Opportunities for group work in lecture, tutorial, assignments
- Grading Scheme
 - Assignments (5): 30%
 - Midterm: 25%
 - Final: 40%
 - Pre-classes quizzes (2x weekly): 2%
 - Clicker: 3%
 - At least obtain 45% weighted average on final and midterm
- Midterm: July 24, Wednesday
- Assignments
 - Released by the end of each week, due the following Friday
 - Can submit in groups of 2-3
 - Drop the lowest assignment grade
 - Best tools for learning and exam preparation (all group members should work on all questions)
- Tutorials
 - Beginning on Friday
 - Wednesday: open office hours, may attend any section
 - Friday: solving practice problems designed to be helpful for following week assignment
- Low-stake components
 - Reading quizzes: three attempts, drop the two lowest quizzes
 - Clicker questions: graded for completion, answer at least 70% of the questions to earn full marks (may opt out, weight shifted, need to email)
- Communication
 - Lectures & Tutorials: in-person, no recording
 - Course materials: Canvas
 - Q & A + Announcements: Piazza
 - Gradescope: assignment submission, linked to CWL

- Logistical concerns: email cp320-admin@cs.ubc.ca (?)
- Sick policy
 - Mild illness: wear mask, limiting close contact
 - Missing lectures: Do the in-class worksheet individually, and then look over the solutions
 - Missing midterm: contact course coordinator (likely to transfer weight to final)
 - Instructor sick (serious): cancel class; (able to teach): online class, announced on Piazza; (non-serious): in-person, wear mask, won't circulate the room during worksheet time
- Piazza
 - At least 4 hours before responding to any Piazza related to course content
 - Aim to resolve all questions within one business day
 - May not receive instructor responses at night, or on weekends and holidays
 - Post content-related questions publicly whenever possible
 - Use private posts for matters that do not concern the rest of the class
 - Keep questions clear and concise
 - Expressing an opinion (challenges in class, issues with students/instructors)? Communicate face to face, via email, or via private post

The Stable Marriage Problem

- Cookies: Golden Oreo (G), Chips Ahoy (C), Double-Stuffed Oreo (D)
Chocolate: Strawberry (S), Coffee (Co), Caramel (Ca)
- Preferences (To trade)

G: Co, Ca, S	S: D, G, C
C: Ca, Co, S	Co: G, D, C
D: S, Co, Ca	Ca: D, C, G
- Possible trades
 - G ---- Co
 - S ---- D
 - Ca ---- C
- Example: Consider an employer-employee list
 - Each employer has a preference ranking of employees
 - Each employee has a preference ranking of employers
 - Want the matching process to be happy (?)
- Trivial and Small instances
 - Trivial Case (Base case in recursion)
 - $n = 0$: do nothing
 - $n = 1$: 1 employer, 1 applicant
 - Easy case: all members have identical preference
 - Small Case

G: Co, Ca, S	S: D, G, C
C: Ca, Co, S	Co: G, D, C
D: S, Co, Ca	Ca: D, C, G

- (S ---- D; G ---- Co; D ---- Ca)
- $e_1: a_1, a_2 ; a_1: e_1, e_2$
- $e_2: a_1, a_2 ; a_2: e_2, e_1$
- Represent the problem (parameter to the solver)
 - Information
 - E = list of employers
 - A = list of applicants
 - $P[e_i]$ = preference list of employer i
 - $P[a_j]$ = preference list of applicant j
 - Validity
 - $|E| = |A|$
 - Preference list for each employer be a permutation (ordering) of the applicants (each applicant appear once and no one else)
 - Preference list for each applicant be a permutation of the employers
- Represent the solution
 - Quantities used
 - $M = \{(e_i, a_j)\}$, a set of employer/applicant pairs
 - Validity
 - $|M| = |E| = |A|$
 - Every employer match with exactly one applicant, no employer/applicant is paired twice (appear in one tuple only)
 - Definition of **good** solution
 - Minimize the number of "bad" matches, give "score of badness"
 - No employer/applicant pair prefers each other over their current match
 - Maximize the total happiness of employers, and disregard applicants; or vice versa
 - The definition we use
 - Intend to be self-enforcing, close to the second definition
 - **Good** solution = no instability
 - Terms
 - Instability: an employer e_i and an applicant a_j , who are NOT matched with each other, but who BOTH prefer each other over their current match
 - Stable matching: a matching that has no instability
 - e.g.
 - $e_1: a_1, a_2 ; a_1: e_1, e_2$
 - $e_2: a_1, a_2 ; a_2: e_2, e_1$
 - An unstable matching would be: $\{(e_1, a_2), (e_2, a_1)\}$
 - Opportunity for e_1 and a_1 to match
 - So the stable matching is: $\{(e_1, a_1), (e_2, a_2)\}$
 - Even though e_2 is not that satisfied, but if e_2 approaches a_1 , a_1 will reject
- Similar problems
- Brute Force?
 - Get all possible solutions and see if one works

- Listing all valid solutions
 - $n!$ matchings
- For each matching, check if there is an instability
 - for each e_i :
 - for each a_j :
 - if e_i and a_j are unmatched && e_i prefers a_j over current partner && a_j prefers e_i over current partner (assume $O(1)$):
 - return True
 - return False
 - $O(n^2)$
- Total complexity: $O(n! \cdot n^2)$
- Promising Approach
 - The Gale-Shapley Algorithm
-

```

Initialize all E, A as unmatched

while there is an unmatched e with at least one $a$
on its preference list:

    Make offer to the next applicant a on e's
    preference list

    If a is unmatched:

        Match e with a

    Else if a prefers e to their current employer
    e':

        Unmatch a and e', and match e with a

    Cross a off of e's preference list

Report the set of matched pairs as the final
matching

```

July 5th - Stable Marriage Problem Reduction

- A group of residents each needs a residency in some hospital. A group of hospitals each need some number (one or more) of residents, with some hospitals needing more and some fewer. Each group has preferences over which member of the other group they'd like to end up with. The total number of slots in hospitals is exactly equal to the total number of residents.
 - We want to fill the hospitals slots with residents in such a way that no resident and hospital that weren't matched up will collude to get around our suggestion (and give the resident a position at that hospital instead).
- Trivial and Small Instances
 - Trivial
 - no hospitals or residents
 - 1 hospital, 1 resident
 - 1 hospital, n residents
 - Small
 - 2 hospitals: 1 hospital (2) - r_1, r_2, r_3 ; $r_1 - h_1, h_2$
 1 hospital (1) - r_1, r_3, r_2 ; $r_2 - h_2, h_1$
 $r_3 - h_1, h_2$
 - $h_1 - r_1, r_2$; $h_2 - r_3$ or $h_1 - r_1, r_3$; $h_2 - r_2$

- 2 hospitals: 1 hospital (2) - r_1, r_2, r_3, r_4 ; $r_1 - h_1, h_2$
1 hospital (2) - r_3, r_4, r_1, r_2 ; $r_2 - h_1, h_2$
 $r_3 - h_2, h_1$
 $r_4 - h_1, h_2$
- $h_1 - r_1, r_2$; $h_2 - r_3, r_4$
- Special case
 - Each hospital has exactly one slot, just typical SMP
- Represent the problem
 - Quantities
 - R : list of residents
 - H : list of hospitals
 - P : preference lists
 - $S(h_j)$: number of slots for hospital j
 - Validity
 - $\#H \geq 1$
 - $\forall j, S(h_j) \geq 1$
 - $\sum_j S(h_j) = |R| = n$
 - all preference lists are permutation of the other side
- Represent the solution
 - Quantities
 - $M = \{(r_i, h_j)\}$
 - Validity and Goodness
 - Valid solution: every r_i appears in exactly one pair and every h_j appears in $S(h_j)$ pairs
 - Good solution
 - An instability occurs when r_i is not assigned to h_j , r_i prefers h_j to their assigned hospital, and h_j prefers r_i to **its least preferred assigned resident**
- Similar problem: SMP
- Brute force
 - **Smallest possible number** of valid solutions for an instance of RHP with n residents is $O(1)$
 - Let n = number of residents, with $S(h_j)$ at disposal
 - The Mississippi problem
 - How many distinct ways are there to rearrange the letters in "Mississippi"?
 - $\frac{11!}{4!4!2!}$
 - So there are $\frac{n!}{\prod_j S(h_j)!}$ distinct solutions
 - Low complexity if some $S(h_j)$ is large
 - If we have many small hospitals, the complexity increases (more realistic)
 - Generating all valid solution
 -
 - Time complexity
 - Check instability: $\leq O(n^2)$
- Promising approach
 - TBC
- Reductions

- Encounter a new problem A, but similar to B
- Definition: an *instance* of a problem is a valid input
- Problem A instance \rightarrow Convert to Problem B instance \rightarrow Problem B solver \rightarrow Convert to Problem A solution
- Time complexity
 - transform problem A to problem B
 - transform B solution to A solution
 - time to solve B
- e.g. SAT: a way to assign truth values to variables such that all clauses evaluate to True
- Promising approach (continued)
 - Small instance reduction
 - 2 hospitals: 1 hospital (2) - r_1, r_2, r_3 ; $r_1 - h_1, h_2$
 1 hospital (1) - r_1, r_3, r_2 ; $r_2 - h_2, h_1$
 $r_3 - h_1, h_2$
 - Treat hospital with $S(h_j)$ slots as $S(h_j)$ hospitals
 - $h_1^1 - r_1, r_2, r_3$; $r_1 - h_1^1, h_2^1, h_2^1$
 $h_1^2 - r_1, r_2, r_3$; $r_2 - h_2^1, h_1^1, h_2^1$
 $h_2^1 - r_1, r_3, r_2$; $r_3 - h_1^1, h_2^1, h_2^1$
 - $h_1^1 - r_1$; $h_1^2 - r_2$; $h_2^1 - r_3$
 - RHP: $\{(h_1, r_1), (h_1, r_2), (h_2, r_3)\}$
 - Algorithm to transform I_{RHP} to I_{SMP}

```

for each r_i in RHP, define r_i in SMP
for each h_j in RHP, define S(h_j) hospitals:
h_j^1, h_j^2, ..., h_j^{S(h_j)} in SMP
P[r_i] in SMP: replace each h_j on RHP
preference by h_j^1, h_j^2, ..., h_j^{S(h_j)}
P[h_j^k] in SMP: = P[h_j]

```
- Algorithm to transform S_{SMP} to S_{RHP}
 - Drop the superscripts in the SMP matching
- Proof of Correctness
 - Assume SMP solver works correctly
 - Want to prove that a stable SMP solution implies a stable RHP solution
 - Easier to prove the contrapositive
 - That is, prove that if RHP solution is not stable, then SMP solution is not stable
 - Suppose the RHP solution has an instability, $\exists r_i, h_j$, s.t. r_i is matched with h_p and h_j is matched with r_q
 r_i prefers h_j over h_p , and h_j prefers r_i over r_q
 Thus, in the SMP solution, r_i is paired with some h_p^m , and h_j^n is paired with some r_q
 $(r_i$ prefers all slots of h_j over all slots of $h_p)$
 therefore, an instability will occur as r_i would prefer h_j^n over h_p^m , and h_j^n prefers r_i over r_q
 this contradicts with SMP solver's correctness
 By contrapositive, if SMP solution is stable, RHP solution is stable
 Q.E.D

July 8th - Asymptotic Analysis

- Asymptotic bounds
 - Compare the growth rate of running times
 - If $\exists c, n_0 > 0$, s.t. $\forall n \geq n_0, f(n) \leq c \cdot g(n)$, we have $f \in O(g)$
 - Conversely, $f \in \Omega(g)$ exactly when $g \in O(f)$
 - $f \in \Theta(g)$ when $f \in O(g)$ and $f \in \Omega(g)$
- Not "best" and "worst" cases
 - O does not mean the worst case, Ω does not mean the best case
 - Best-case, worst-case are properties of an algorithm
 - worst-case for QuickSort with first element chosen as pivot is a list that's already sorted (gives $\Theta(n^2)$ runtime)
 - The two notations are ways to compare functions and don't inherently have anything to do with algorithms
- Little o and ω (strictly slower or faster)
 - $\forall c \in \mathbb{R}^+, \exists n_0 \in \mathbb{Z}^+$ such that, $\forall n \geq n_0, f(n) \leq c \cdot g(n)$, then we say $f \in o(g)$
 - e.g. $2n \in o(n^2)$, but $2n \notin o(n)$ (because we do not have $2n \leq cn$, for say $c = 0.1$)
 - Conversely, $f \in \omega(g)$ exactly when $g \in o(f)$
 - Definitions via limits

- $$\lim_{n \rightarrow \infty} \frac{f}{g} = \infty \rightarrow g \in o(f), f \in \omega(g)$$

- $$\lim_{n \rightarrow \infty} \frac{f}{g} = 0 \rightarrow f \in o(g), g \in \omega(f)$$

- $$\lim_{n \rightarrow \infty} \frac{f}{g} = c \rightarrow f \in \Theta(g), g \in \Theta(f)$$

- Comparing Orders of Growth for Functions
 - Give the best Θ bound and arrange by increasing order of growth
 - $n + n^2 \in \Theta(n^2)$
 - $2^n \in \Theta(2^n)$
 - $55n + 4 \in \Theta(n)$
 - $1.5n \lg n \in \Theta(n \log n)$
 - $n! \in \Theta(n!)$
 - $\ln n \in \Theta(\log n)$
 - $2n \log(n^2) \in \Theta(n \log n)$
 - $\frac{n}{\log n} \in \Theta(\frac{n}{\log n})$
 - $(n \lg n)(n + 1) \in \Theta(n^2 \log n)$
 - $(n + 1)! \in \Theta((n + 1)!)$
 - $1.6^{2n} \in \Theta(2.56^n)$
 - $\Theta(\log n) < \Theta(\frac{n}{\log n}) < \Theta(n) < \Theta(n \log n) < \Theta(n^2) < \Theta(n^2 \log n) < \Theta(2^n) < \Theta(2.56^n) < \Theta(n!) < \Theta((n + 1)!)$
 - $\lim_{n \rightarrow \infty} \frac{(n + 1)!}{n!} = \lim_{n \rightarrow \infty} (n + 1) = \infty$
 - $\lim_{n \rightarrow \infty} \frac{2.56^n}{2^n} = \lim_{n \rightarrow \infty} (\frac{2.56}{2})^n = \infty$
 - $\lim_{n \rightarrow \infty} \frac{\frac{n}{\log n}}{\log n} = \lim_{n \rightarrow \infty} \frac{n}{(\log n)^2} = \lim_{n \rightarrow \infty} \frac{1}{2 \log n \cdot \frac{1}{n}} = \lim_{n \rightarrow \infty} \frac{n}{2 \log n} = \lim_{n \rightarrow \infty} \frac{1}{2 \cdot \frac{1}{n}} = \lim_{n \rightarrow \infty} \frac{n}{2} = \infty$
- Functions / Orders of Growth for Code

- ```

Let max = -infinity
for each element a in A:
 If max < a:
 Set max to a
Return max

```

- $\Theta(n)$

- ```

Let first = A[1]
Let last = A[n]
Let middle = A[floor(n/2)]

If first <= middle and middle <= last:
    return middle
Else if middle <= first and first <= last:
    return first
Else:
    return last

```

- $\Theta(1)$

- ```

Let inversions = 0
for each index i from 1 to n:
 for each index j from (i+1) to n:
 If a[i] > a[j]:
 Increment inversions
Return inversions

```

- $\Theta(n^2)$

- $i = 1, j = 2 \text{ to } n: n - 1$

- $i = 2, j = 3 \text{ to } n: n - 2$

- $\vdots$

- $(n - 1) + (n - 2) + \dots + 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$

- Progress Measures for While Loops

- ```

Let i = FindNeighboringInversion(A)
while i >= 0:
    Swap A[i] and A[i + 1]
    Set i to FindNeighboringInversion(A)

```

- **FindNeighboringInversion(A)** consumes an array **A** and returns an index **i** such that **A[i] > A[i+1]** or returns **-1** if no such inversion exists

- Two small inputs

- $[4, 3, 2, 1]$ stresses the algorithm

- $\rightarrow [3, 4, 2, 1] \rightarrow [3, 2, 4, 1] \rightarrow [2, 3, 4, 1] \rightarrow [2, 3, 1, 4] \rightarrow [2, 1, 3, 4] \rightarrow [1, 2, 3, 4]$

- $[1, 2, 3, 4]$ easiest

- $[3, 2, 4, 1]$ common

- Define an inversion, and prove that if an inversion exists, a neighboring inversion exists

- Inversion: For $i < j$, inversion is where $A[i] > A[j]$

- Proof: Suppose \exists an inversion between element **i** and element **i + k** where $(1 \leq k \leq n - 1)$, prove by induction on **k** that there is a neighboring inversion in **[i, i+k]**

Base case: $k = 1$, $A[i] > A[i+1]$, which is a neighboring inversion

Induction hypothesis: Assume \exists a neighboring inversion between any inverted elements that are $k - 1$ apart

Inductive step: prove for a gap size k , assuming $A[i] > A[i+k]$,
,

if $A[i+k-1] > A[i]$, then $A[i+k-1] > A[i+k]$, a neighboring inversion;

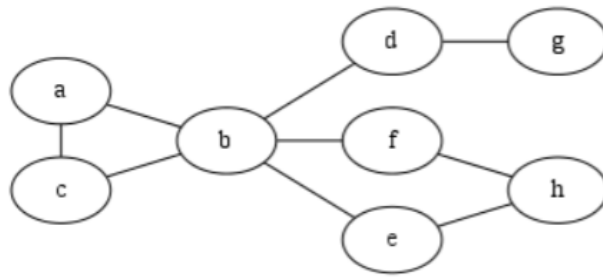
if $A[i+k-1] < A[i]$, then by IH, there exists a neighboring inversion within the $k - 1$ gap, which is also within the k gap.

□

- Upper and lower bounds on the number of inversions in A
 - lower bound: 0
 - upper bound: nC_2
- Give a "measure of progress" for each iteration of the loop in terms of inversions (how do we know we are terminating the loop)
 - number of neighboring inversions? does not decrease in **every** step
 - total number of inversions? always decreases by 1 **each** step, fixing a neighboring inversion removes exactly 1 inversion (at most 1, because elements outside the neighboring pair do not get moved)
- Upper-bound on the number of iterations the loop could take
 - max number of iterations = total number of inversions
 - the number of inversions goes down by 1 at each loop iteration, and terminates when there are no inversions
- Prove that this algorithm sorts the array A (i.e. removes all inversions from the array)
 - There are 1 iteration per array inversion (by "measure of progress")
 - There are at most nC_2 inversions/iterations
 - Algorithm terminates when there are no neighboring inversions \implies no inversions, by contrapositive

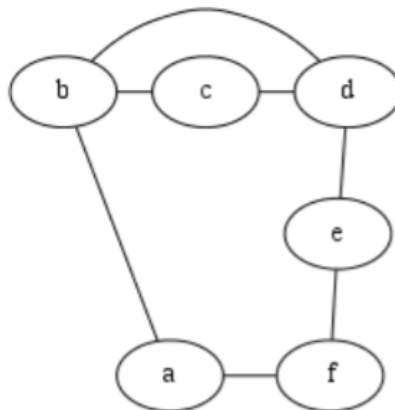
July 10th - Graphs

- Terms
 - Articulation point: a vertex in an undirected graph whose removal increases the number of connected components in the graph.
 - Diameter: in an undirected, unweighted graph, it is the largest possible value of the smallest number of edges on any path between two nodes; the largest number of steps required to get between two nodes in the graph.
- Play

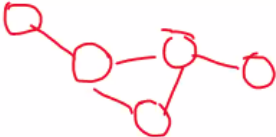


- Articulation points: b, d
 - Removing b, would yield 3 connected components
 - Removing d, would yield 2 connected components
- Diameter: 4
 - g → h
 - The largest possible steps required (farthest pair of points) for the shortest path (the shortest number of edges required to traverse these two points)
- BFS
 - a → b, a → c
 - b → d, b → e, b → f, b → h
 - d → g
 - e → h
 - f → h

- DFS
 - a → b → c, a → h
 - b → d → g, b → e → h
 - b → f, f → h



- Articulation point: None
- Diameter: 3
 - c → f
- BFS
 - a → b, a → f
 - b → c, b → d
 - c → d, d → e
- DFS
 - a → b → c → d → e → f
 - a → f, b → d

- Diameter Algorithm
 - Design an algorithm to find the **diameter** of an unweighted, undirected, **connected** graph. Call this problem DIAM
 - Trivial and Small Instances
 - Trivial
 - 1 vertex - diameter = 0
 - 2 vertices - diameter = 1
 - 0 vertices - diameter = ...?
 - Easy case: "star"/"pinwheel" vertex, straight line
 - Small
 - "pinwheel" - all vertices connected to one single center vertex
 - "chain" - all vertices align in a chain
 - 
- Represent the problem
 - Expression
 - $G = (V, E)$
 - Constraints
 - $\forall u \in V, (u, u) \notin E$
 - All edges in E are unique, if $(u, v) \in E$, then $(v, u) \notin E$
- Represent the solution
 - Output
 - an integer, d
 - diametric nodes u and v (easy to check all possible diametric nodes)
 - Validity
 - d should be non-negative; d should be the distance from u to v
 - Goodness
 - should actually be the diameter (i.e. d is the length of a path between two nodes, and is the shortest of all such paths)
 - d is the longest such distance, u and v are the diametric nodes
- Similar problems
 - Dijkstra???
- Brute Force
 - Sketch an algorithm
 - ```
max = 0
for u in V:
 for v in V:
 d = length of path from u to v
 found by BFS
 if d > max:
 max = d
```

- Appropriate variable to represent the "size" of an instance
  - $n = |V|, m = |E|$
- How many such "solution form" are there
  - $O(n^2)$ :  $n^2$  for our particular algorithm in "sketch an algorithm" but  $nC_2$  for a better approach
- What will characterize how good a possible solution is?
  - is it the highest  $d$  value that we have seen so far
- Determine how good it is?
  - Check the value of  $d$ .
- Will brute force be sufficient
  - Runtime:  $O(n^2)$  (all pairs)  $\times O(m + n)$  (BFS)  
 $= O(n^3 + mn^2)$
  - If graph is sparse:  $O(n^3)$
  - If graph is dense:  $O(n^4)$
- Promising approach
  - Tweak the brute force algorithm
    - BFS is run in each loop, too many repetitions
    - ```

max = 0
for u in v:
    Run BFS from u (until entire graph
    is explored)
    Find the node v with maximum depth
    in the tree
    if d > max:
        max = d
        diamNodes = (u, v)
return max, diamNodes
              
```
- Runtime
 - for loop: $O(n)$
 - BFS: $O(m + n)$
 - Find node v : $O(1)$
 - Conditional: $O(1)$
 - Return: $O(1)$
 - Total: $O(n(n + m)) = O(n^2 + mn)$

July 12th - Greedy Algorithm

- Optimization problems
 - We have a problem with several valid solutions
 - There is an **objective function** that tells us how good/bad each solution is
 - We are looking for the valid solution that minimizes or maximizes the value of the objective function
- Defining greedy algorithms
 - It proceeds by
 - making a choice based on a simple, local criterion
 - solving the subproblem that results from that choice
 - combining the choice and the subproblem choice

- making a **sequence of (locally "good") choices**
- no precise definition of "greedy"
- e.g. choosing the interval with the earliest finish time
 - choosing the job with the earliest deadline
 - choosing the item needed further in the future
 - choosing the smallest weight edge that does not create a cycle
- Does it always give the correct solution
 - sometimes yes, sometimes no, generally no
 - there are some classes of problems for which there exists a greedy algorithm that always returns the correct solution
 - there are problems where no one knows any greedy algorithm with this property
 - there are some problems where greedy doesn't give an optimal solution, but it's provably **close** to optimal in some sense
- Method 1: Greedy stays ahead
 - ??? (went to write some homework, should not start that late next time)
- Method 2: exchange arguments
 - Prove that if O is an optimal solution, and G is the greedy solution, then we can modify O slightly to get O' such that:
 - O' is more similar to G than O was
 - O' is at least as good a solution as O
 - Then describe how you can repeatedly modify O until it is the *same* solution as G , without ever decreasing the solution quality
 - "more similar"
 - has more edges in common with
 - selects more of the same jobs
 - has fewer elements out of order compared with the greedy solution
- Worked examples (greedy stays ahead)
 - Rules
 - each TA can carry no more than m exams
 - The exams are expected at the CS building in a particular order, so they need to be transported back in the order listed (bundle 1, ..., bundle n)
 - Can't split any of the exam bundles among multiple TAs, since some exams may end up getting misplaced
 - Distribute the exams to TAs such that you transport the exams using as few TAs as possible
 - Optimal greedy algorithm: give each TA as many exam booklets as they can carry before assigning any booklets to the next TA
 - We want a **greedy stays ahead lemma**
 - should be inductive
 - should imply (if true) that the greedy algorithm is optimal
 - should actually be true for our algorithm
 - The first i TAs in the greedy solution are carrying at least as many total booklets as the first i TAs in the optimal solution

- Suppose the greedy solution uses k TAs to transport all $\sum b_i$ exams. Optimal cannot use $j < k$ TAs, because the first j TAs in greedy are transporting fewer than $\sum b_i$ exams, and lemma tells us that the first j TAs in optimal can't be transporting more than that. Therefore, greedy uses no more TAs than optimal, and is thus optimal.
- Worked example (exchange argument)
 - Mr. Plow has n clients, each job takes 1 hour to complete. All of the clients would prefer to have theirs done earlier rather than later. If client i is Mr. Plow's j th client, then client i 's satisfaction is $s_i = n - j$. Some clients are more important, with i th client having weight w_i . We want to maximize the satisfaction.
 - Optimal: decreasing w_i
 - Exchange argument
 - Let O denote an optimal solution (o_1, o_2, \dots, o_n) different from our greedy solution G
 - need to define a way to make a single "exchange" to O to make it "closer" to G , we define a single "difference" as an inversion
 - Consider $p < q$ where $w_p \leq w_q$. We show we can swap o_p and o_q and get a solution with larger/equal weighted satisfaction
 - Repeatedly swap inversions until the optimal solution becomes the greedy solution, all without decreasing weighted satisfaction
- Clustering (part 1)
 - Build intuition through examples
 - Data structure: undirected weighted graph
 - Trivial
 - 1 photo
 - 0 photos
 - $K = 1$ for any number of photos
 - n photos, $K = n$ categories
 - Develop a formal problem specification
 - Develop notation for describing a problem instance
 - $G = (V, E)$, $e = (u, v, w)$
 - number of categories k
 - Use your notation to flesh out the following group of photos into an instance
 - $(v_1, v_2, 0.7), (v_1, v_3, 0.9), (v_3, v_4, 0.4), (v_4, v_2, 0.3), (v_1, v_4, 0.2), (v_2, v_3, 0.2)$
 - $k = 2$
 - Develop notation for describing a potential solution
 - C_1 = set of photos in category 1, C_k = set of photos in category k
 - Define a good solution
 - Maximal similarity in each category: objective function is the maximum similarity between any two photos in any category; and we want to maximize this
 - Average of all edges between all pairs in the set
 - Sum of similarities for each category
 - General approaches

- we can look at edges within categories (intra-category edges), these are ones that we want to be higher
 - Sums, averages, max or min (different "vector norms")
- we can look at edges between categories (inter-category edges), these are ones that we want to be lower
 - sum, averages, etc; can subtract from the intra-category values
- Definition of goodness that we follow
 - Define the similarity between two categories C_i and C_j to be the maximum similarity between any pair of photos p_m, p_n such that $p_m \in C_i$ and $p_n \in C_j$
 - Cost of a categorization is the maximum similarity between any two of its categories. Lower the cost, better the categorization.
 - Cost of solution = the weight of the heaviest inter-category edge
- Similar problem
 - Kruskal's Minimum spanning tree
- Brute force
 - Potential solution is the set of partitions of n photos into c subsets. Number of potential solutions?
 - If $c = 2$, $(2^n - 2)/2 = 2^{n-1} - 1$
 - Determine how good it is? Asymptotically
 - $O(n^2)$
- Design a better algorithm
 - Find the edge in each of your instances with the highest similarity. Should the two photos incident on that edge go in the same category?
 - They should go in the same category. If not, that edge weight will define our cost, and the cost will be **as bad as possible** for that instance (any categorizations where these photos **are not** in the same category will all be equally and maximally bad)
 - Algorithmic idea
 - Greedy algorithm is to initialize all photos into their own categories, and then merge categories of the two photos with max edge weight, until we reach the desired number of categories.
 - ```

Initialize each photo as its own
category
n_c = n
while n_c > k:
 find the two photos u, v with max
 edge weight e
 if u, v are in different
 categories:
 Merge the categories of u and
 v
 n_c -= 1
return the categorization

```

## July 15th - Clustering Completed

- Description
  - We are given a complete, weighted, undirected graph  $G = (V, E)$  represented as an adjacency list, where the weights are all between 0 and 1 and represent similarities - the higher the more similar - and a desired number  $1 \leq k \leq |V|$  of categories
  - We define the similarity between two categories  $C_1$  and  $C_2$  to be the maximum similarity between any pair of nodes  $p_1 \in C_1$  and  $p_2 \in C_2$ . We must produce the categorization - partition into  $k$  (non-empty) sets - that minimizes the maximum similarity between categories
  - Objective function: weight of max-weight edge between any two categories
- Algorithmic approach
  - Sort a list of the edges  $E$  in decreasing order by similarity
  - Initialize each node as its own category
  - Initialize the category count to  $|V|$
  - While we have more than  $k$  categories
    - Remove the highest similarity edge  $(u, v)$  from the list
    - If  $u$  and  $v$  are not in the same category: merge  $u$ 's and  $v$ 's categories, and reduce the category count by 1
  - The highest weight edge in the graph needs to be "inside" a single category to avoid an **optimally bad** solution
  - At subsequent steps, we want the next highest-weight inter-category edge to be placed in a single category or merge into an existing category
- Greedy is at least as good as Optimal
  - Can a greedy solution  $G$  have an intra-category edge with lower weight than  $e$  (inter-category edge with maximum similarity  $e$ )
    - Yes
    - The intra-category edges on one hand can be visited by its edge weight, but can also be visited when two categories merge (which are not ranked by edge weight)
  - Give a lower/upper bound on the maximum similarity of an arbitrary categorization  $C$  in terms of any one of its inter-category edge weights. If  $C$  has an inter-category edge with weight  $s$ , how much can be told about  $\text{Cost}(C)$ 
    - $\text{Cost}(C) \geq s$
  - Case 1: a blue edge (intra-category edge in  $G$ ) is inter-category in  $O$ 
    - Let  $G$  be the categorization produced by the greedy algorithm, and let  $O$  be an optimal categorization. Let  $E'$  be set of edges removed from the list during iterations of the **while** loop. w.r.t the greedy solution  $G$ , are the edges in  $E'$  inter-category? Or intra-category? Or could be both type?
      - Greedy algorithm merges on these edges if they are not intra-category already, thus, all edges contained by  $E'$  are intra-category
    - Suppose that some edge  $e = (p, p', s)$  of  $E'$  is inter-category in the optimal solution  $O$ , what can we say about  $\text{Cost}(G)$  vs.  $\text{Cost}(O)$ 
      - $\text{Cost}(G) \leq s$ , because Greedy merges on max-weight edges, so the leftover inter-category edges must have weight  $\leq s$
      - From above  $\text{Cost}(O) \geq s$



- So  $\text{Cost}(G) \leq s \leq \text{Cost}(O)$
- Case 2: All blue edges (intra-category edge in  $G$ ) are intra-category in  $O$ 
  - Suppose all edges of  $E'$  are intra-category not only in  $G$ , but also in the optimal solution  $O$ . Can there be any edges that are inter-category in  $G$  but intra-category in  $O$ ? Can you convert any of  $G$ 's inter-category edges to intra-category edges without either making some edges in  $E'$  inter-category or making your solution invalid?
    - There cannot be any edges that are inter-category in  $G$  but intra-category in  $O$
    - Suppose we have all edges in  $E'$  are intra-category in  $O$ , and suppose  $(u, v)$  is inter-category in  $G$ , for  $(u, v)$  to be intra-category in  $O$ 
      - We either merge the categories of  $u$  and  $v$  in  $O$  implying  $O$  has fewer categories than  $G$ , which is not allowed
      - Or we can move  $v$  to  $u$ 's category, this means either "breaking"  $v$  away from one or more nodes in its component, implying one of the edges in  $E'$  will become inter-category, or merge  $v$ 's category, which is not allowed
  - Slightly stronger result: If all  $E'$  edges are intra-category in  $O$ 
    - all inter-category edges in  $G$  are inter-category in  $O$
    - $E'$  edges are intra-category in both  $G$  and  $O$  (by assumption in Case 2)
    - all intra-category non- $E'$  edges in  $G$  are also intra-category in  $O$  (because they are incident on two vertices in the same merged component)
    - $G$  is  $O$
- Conclude that  $G$  must be an optimal solution
  - Suppose we have an optimal solution  $O \neq G$
  - So Case 1: some edge in  $E'$  is inter-category in  $O$ , by above  $\text{Cost}(G) \leq \text{Cost}(O)$
  - Case 2: all edges in  $E'$  is intra-category in  $O$ , by above  $G = O$ , contradicts with supposition, implying  $\text{Cost}(G) = \text{Cost}(O)$
  - In all cases:  $\text{Cost}(G) \leq \text{Cost}(O)$ , implying  $G$  is optimal

## July 17th - Divide and Conquer

- Divide and Conquer Algorithm
  - Definitions
    - Dividing the input into 2+ smaller instances of the same problems - we call these **subproblems**
    - Solving the subproblems recursively
    - Combining the subproblem solutions to obtain a solution to the original problem
    - We will also consider **prune and search** algorithms, which look for elements with a specific property and only recurse on one subproblem
  - Classic examples

- QuickSort, MergeSort
- Binary Search (prune and search)
- Recurrence relations
  - The running time  $T(n)$  of a recursive function can be described using a recurrence relation: it is defined in terms of one or more terms of the form  $T(m)$  ( $m < n$ )
  - $$T(n) = \begin{cases} T(n/2) + 2T(n/4) + n^2 & n \geq 4 \\ \Theta(1) & n \leq 3 \end{cases}$$
- Recursion Trees
  - Represent the recursion with a tree where each node represents a recursive subproblem
  - Inside each node, write the size of the subproblem this call to the function solves
  - Next to each node, write the amount of work done by the call to the function, **not including** any time spent in recursive calls
  - Compute the total amount of non-recursive work done on each row
  - Then add up the work done over all rows
- The Master Theorem
  - Most recursion trees fall into one of 3 categories
    - work per level increases geometrically
    - work per level is constant
    - work per level decreases geometrically
  - Let  $a \geq 1, b > 1$  be real constants, let  $f: \mathbb{N} \rightarrow \mathbb{R}^+$  and let  $T(n)$  be defined by

$$T(n) = \begin{cases} aT(n/b) + f(n) & n \geq n_0 \\ \Theta(1) & n < n_0 \end{cases}$$

where  $\frac{n}{b}$  might be either  $\text{ceil}(n/b)$  or  $\text{floor}(n/b)$

Then:

- If  $f(n) \in O(n^{\log_b a - \epsilon})$  for some  $\epsilon > 0$  then  
 $T(n) \in \Theta(n^{\log_b a})$
  - If  $f(n) \in \Theta(n^{\log_b a} \log^k n)$  for some  $k \geq 0$  then  
 $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$
  - If  $f(n) \in \Omega(n^{\log_b a + \epsilon})$  for some  $\epsilon > 0$  and  
 $af(n/b) < \delta f(n)$  for some  $0 < \delta < 1$  (regularity condition) and all  $n$  large enough, then  
 $T(n) \in \Theta(f(n))$
- Application
    - Compute  $\log_b a$
    - Compare it to the exponent of  $n$  in  $f(n)$ 
      - If  $\log_b a$  is larger: Case 1
      - If  $\log_b a$  and the exponent are equal: maybe Case 2
      - If  $\log_b a$  is smaller: check regularity condition, and if it holds then Case 3
  - Examples
    - $T(n) = 5T(\sqrt{n}) + n$ 
      - Cannot apply
    - $T(n) = T(n/2) + 1$ 
      - $a = 1, b = 2, \log_b a = 0$

- $n^0 = 1$
- $1 \in \Theta(n^0 \log^0 n) \rightarrow T(n) \in \Theta(n^0 \log n) = \Theta(\log n)$
- $T(n) = T(n/4) + n$ 
  - $a = 1, b = 4, \log_b a = 0$
  - $n^1 = n$
  - $af(n/b) = f(n/4) = n/4 < \delta n (\delta = \frac{1}{2})$
  - $T(n) \in \Theta(n)$
- $T(n) = 3T(n/9) + \sqrt{n} \log_2 n$ 
  - $a = 3, b = 9, \log_b a = \frac{1}{2}$
  - $\sqrt{n} \log_2 n = n^{\frac{1}{2}} \log_2 n$
  - $\sqrt{n} \log_2 n \in \Theta(n^{\frac{1}{2}} \log n) \rightarrow T(n) \in \Theta(\sqrt{n} \log^2 n)$
- $T(n) = \sqrt{n} T(n/3) + n^2$ 
  - Cannot apply
- $T(n) = 9T(n/3) + n \log n$ 
  - $a = 9, b = 3, \log_b a = 2$
  - $T(n) \in \Theta(n^2)$
- $T(n) = 2T(n/2) + \frac{n}{\log n}$ 
  - $a = 2, b = 2, \log_b a = 1$
  - $n = n^1$
  - $k = -1$  for  $\frac{n}{\log n}$ , cannot apply
  - Can we check for Case 1? If it is possible, we want  $\frac{n}{\log n} \in O(n^{1-\epsilon})$ , which is we want  $n^\epsilon \in O(\log n)$ , which is not possible
- QuickSort Runtime Analysis
  - Recurrence relation
    - $T(n) = T(n/4) + T(3n/4) + cn$
  - The number of levels in the tree down to the shallowest leaf
    - depth  $i$ : subproblem of size  $\frac{n}{4^i} = 1 \rightarrow i = \log_4 n$
  - The number of levels in the tree down to the deepest leaf
    - depth  $i$ : subproblem of size  $\frac{3^i}{4^i} n = 1 \rightarrow i = \log_{\frac{4}{3}} n$
  - Find asymptotic upper and lower bounds for the solution of your recurrence
    - $T(n) \in \Omega(cn \cdot \log_4 n)$
    - $T(n) \in O(cn \cdot \log_{\frac{4}{3}} n)$
    - $T(n) \in \Theta(n \log n)$
  - Now consider a weaker assumption that the rank of the pivot is always in the range between  $\frac{n}{4}$  and  $\frac{3n}{4}$ . What can you say about the running time of QuickSort in this case?
    - Pivots at the ends of the array are bad (can be  $n^2$ )
    - Pivots in the middle of the array are good (average case is between  $n/4$  and  $3n/4$ )
    - The recursion tree will be more balanced (shallowest leaf will be deeper, and deepest leaf will be shallower)
    - Let  $\frac{1}{4} < k < \frac{3}{4}$ , then the lower bound will be  $n \cdot \log_{\frac{1}{k}} n \geq n \cdot \log_4 n$ , the upper bound will be  $n \cdot \log_{\frac{1}{1-k}} n \leq n \cdot \log_{\frac{4}{3}} n$
    - As long as we have a  $0 < k < 1$  and we choose  $kn$  and  $(1-k)n$ , it remains  $n \cdot \log n$

## July 19th - Divide and Conquer

- The Stock Market, dividends, and risks
  - Brute Force
    - Takes  $O(n^2)$  time, pairwise comparison between points
  - Describe an algorithm that returns all points of  $P$  that  $q$  does not dominate

```
AtLeastGoodAsQ(P, q):
 Let L be a list of points that q does not
 dominate
 For each p in P:
 If p.x > q.x or p.y > q.y
 Append p to L
 Return L
```

- How many subproblems should we use, and how is the input divided between these subproblems
  - 2 subproblems
  - Left half and right half based on  $x$  coordinates
- If a point is maximal in the right of the subproblems, is it automatically maximal in  $P$ ?
  - It is maximal in  $P$ , the only points not considered in this half are entirely to the left, thus they will not dominate any points in the right half.
- If a point is maximal in the left of the subproblems, is it automatically maximal in  $P$ ?
  - It is not always maximal in  $P$ . All points on the right dominate the  $x$ -coordinate, there can exist one point that has a larger  $y$ -coordinate, dominating the maximal point in the left half.
- Suppose that a point  $p$  in the left subproblem is dominated by one or more points in the other sub-problem. Name *one* point in the other subproblem that is guaranteed to dominate  $p$  (?).
  - Compare each maximal point in the left problem with the point on the right with the largest  $y$  coordinate
- Write pseudo-code for an efficient algorithm **MaximalPoints** that takes an input a set of  $P$  points, and return a set of all maximal points of  $P$

```
MaximalPoints(P):
 Let S be the set for such maximal points
 if P has 1 point i:
 Append i to S
 else:
 Let x_max be the maximum x coordinate
 of all points in P, initialized to 0
 For each point p in P:
 if p.x >= x_max:
 x_max = p.x
 Let P_L, P_R be the left half and
 right half of points in P respectively
 For each point p in P:
 if p.x <= x_max / 2:
 Append p to P_L
 else
 Append p to P_R
 Let S_L = MaximalPoints(P_L)
 Let S_R = MaximalPoints(P_R)
```

- ```

    Let y_max = -10
    For each p in S_R:
        if p.y >= y_max:
            y_max = p.y
    Append p in S
    For each p in S_L:
        if p.y > y_max:
            Append p in S

    Return S

```
- Sort P by x-coordinate


```

MaximalPoints(P):
    if |P| <= 1:
        return P

    LeftHalf = P[1: floor(|P|/2)]
    RightHalf = P[floor(|P|/2)+1 : end]
    MaxPointsL = MaximalPoints(LeftHalf)
    MaxPointsR = MaximalPoints(RightHalf)
    Let q = point in RightHalf with largest
    y-coordinate
    Return NonDomPoints(MaxPointsL, q) +
    MaxPointsR
      
```
 - Analyze the runtime of algorithm
 - $O(n \log n) + [O(1) + 2T(n/2) + O(n)]$
 - $T(n) = 2T(N/2) + O(n) \rightarrow T(n) = O(n \log n)$
 - Prune and Search
 - An algorithm for finding the median
 - Use QuickSort to find the median of an array. Assume that QuickSort: Selects the first element as pivot and maintains elements' relative order when producing **Lesser** and **Greater**, for list [10, 3, 5, 18, 1000, 2, 100, 11, 14]
 - [3, 5, 2] [10] [18, 1000, 100, 11, 14]
 - [[2] [3] [5]] [10] [[11 14] [18] [1000 100]]
 - [[2] [3] [5]] [10] [[11] [14] [18] [100] [1000]]
 - Look at the **third** recursive call you marked. What is the rank of 11 in this array? How does this relate to 11's rank in the second recursive call, and why?
 - It is ranked 1 in this array, and ranked the same in the previous recursive call
 - Because 11 went to **Lesser**, so all "dropped" elements are all larger elements (**pivot** + **Greater**)
 - Now, look at the second recursive call
 - In this array, the median is 18
 - The rank of the median is 3
 - The rank of 11 is 1
 - In the original array, the rank of 11 is 5, it has gone down by 4 because the original array dropped pivot, and **Lesser** (it ends up in **Greater**); it goes down the number of elements in **Lesser** + 1.
 - When 11 goes to **Lesser**, the rank does not change; when 11 goes to **Greater**, the rank decreases
 - If you are looking for the element of rank 42 in an array of 100 elements, and **Lesser** has 41 elements, where is the element you are looking for?
 - It is the pivot

- How could you determine **before** making **QuickSort**'s recursive calls whether the element of rank k is the pivot or appears in **Lesser** or **Greater**?
 - If **Lesser** has $k - 1$ elements \rightarrow rank k element is pivot
 - If **Lesser** has at least k elements \rightarrow rank k element is in **Lesser**
 - If **Lesser** has no more than $k - 2$ elements \rightarrow rank k element is in **Greater**
- Modify the **QuickSort** algorithm so that it finds the element of rank k

```
function QuickSelect(A[1...n], k):
    if n > 1 then:
        choose pivot element p = A[1]
        Let Lesser be elements < p
        Let Greater be elements > p
        if k = |Lesser| + 1:
            (|Lesser| == k - 1)
                return p
            else if k <= |Lesser|:
                (|Lesser| > k - 1)
                    return QuickSelect(Lesser,
k)
            else:
                (|Lesser| < k - 1)
                    return
                QuickSelect(Greater, k - |Lesser| - 1)
            else:
                return A
```

- Suppose that the rank of the pivot in median-finding algorithm on size n is always in range between $\lceil \frac{n}{4} \rceil$ and $\lfloor \frac{3n}{4} \rfloor$. Give a tight O bound on the running time. Also provide an asymptotic lower bound on the algo's running time.
 - Worst case is we check the list of $\frac{3n}{4}$ at each recursive call (it's always checking **Greater**)
 - $n + \frac{3n}{4} + \frac{9n}{16} + \dots \in \Theta(n)$

July 22nd - Prune and Search

- Deterministic Select (Avoid $\Theta(n^2)$, since sorting and taking the middle element is quicker asymptotically)
 - Assume you can magically choose a pivot, but you cannot force the pivot to always be the element you're searching for, what to do?
 - Goal: there should be some $\delta < 1$ such that the size of **Lesser**, **Greater** $< \delta n$
 - Since you don't have magic, we need to choose pivot in the old-fashioned way, suppose an array
 - 19, 35, 44, 13, 42, 23, 21, 62, 27, 24, 40, 53, 31, 16, 48, 25, 17, 13, 9, 26, 18, 50, 21, 30, 67, 17, 71, 87
 - Group them in groups of 5, circle median of each group
 - [19, 35, 44, 13, 42], [23, 21, 62, 27, 24], [40, 53, 31, 16, 48], [25, 17, 13, 9, 26], [18, 50, 21, 30, 67], [17, 71, 87]
 - 35, 24, 40, 17, 30, 71

- What is the median of the circled elements?
 - 32.5 (30)
- How many of the elements are smaller than that median? How many are larger?
 - 17 are smaller, 11 are larger (15 are smaller, 12 are larger)
- Generalize the previous process, split the array in groups of 5, find the median of each group and find the median of medians to be m , how many groups of 5 will we get?
 - $\lceil \frac{n}{5} \rceil$
- How many elements of the array are *guaranteed* to be smaller than m ? What about the number of elements are *guaranteed* to be larger than m ?
 - $\frac{3n}{10}$ *guaranteed* to be smaller, $\frac{3n}{10}$ *guaranteed* to be smaller
- What is the *largest* possible size of one of **Lesser**, **Greater** lists
 - Choose p (pivot) as follows divide our input into groups of 5, find median of each group of 5 and set p to be the median of the $\lceil \frac{n}{5} \rceil$ group medians
 - $\frac{3n}{10} + \frac{4n}{10} = \frac{7n}{10}$ to the largest case
- Derive a recurrence relation for the worst-case running time $T(n)$ of algorithm **DeterministicSelect** running on an array with n elements
 - $T(n) \leq T(\frac{7n}{10}) + T(\frac{n}{5}) + cn$, $T(\frac{n}{5})$ is the running time to find the median of group medians, cn is the running time to divide groups of 5 + find the median of each of these group (each group takes $O(n)$)
- Prove that the function $T(n) \in \Theta(n)$
 - First level: $\frac{7n}{10}, \frac{2n}{10}$
 - Second level: $\frac{49n}{100}, \frac{14n}{100}, \frac{14n}{100}, \frac{4n}{100}$
 - $1 + \frac{9n}{10} + \frac{81n}{100} + \dots = c$

Exam Review

- $f(n) \in \Theta(g(n))$
 - True: $g(n) \in \Theta(f(n) + g(n))$, $\sqrt{f(n)} \in \sqrt{g(n)}$
 - False: $2^{f(n)} \in \Theta(2^{g(n)})$
- Assignment 2 Q4
 - Look at interval of earliest end time, choose the interval that overlaps with it with the latest end time
 - Greedy stays ahead lemma must have
 - about partial solutions
 - should imply that greedy is optimal
 - should be true about our algorithm
 - Lemma
 - the first i shifts in the greedy solutions can cover as least many shifts as the first i shifts in the optimal solution;
 - the i th shift in greedy ends no earlier than the i th shift in the optimal solution and every earlier shift is covered

July 26th - Dynamic Programming and Memoization

- Build intuition through examples
- Writing down a formal problem specification
- Evaluating brute force solution

```
function Brute-Force-Change(n):  
    if n < 0:  
        return infinity  
    else if n = 0:  
        return 0  
    else:  
        return min(1 + BFC(n - 25),  
                    1 + BFC(n - 10),  
                    1 + BFC(n - 1))
```

$$T(n) = \begin{cases} c_1 & n \leq 0 \\ T(n-25) + T(n-10) + T(n-1) + 1 & \text{otherwise} \end{cases}$$

- Give a Ω bound
 - $T(n) \geq 3T(n-25) + 1$
 - At level i : 3^i ; There are $\frac{n}{25}$ levels
 - So the least amount of total work is $1 + 3 + 9 + \dots + 3^{\frac{n}{25}}$
 - So $T(n) \in \Omega(3^{\frac{n}{25}})$
 - If we give an upper bound, we thus have $T(n) \in O(3^n)$
- Are we solving the same subproblem repeatedly?
 - Given $n = 81$, on level 2, we are solving $BFC(46), BFC(55), BFC(70)$ multiple times
 - Number of nodes in the tree $\Omega(3^{\frac{n}{25}}), O(3^n)$
 - Number of unique nodes is $O(n)$
 - Exponential repetition of nodes in the recursion tree
- Dynamic Programming
 - A dynamic programming proceeds by
 - making a choice
 - solving the subproblem that results from that choice
 - combining the choice and the subproblem choice
 - This technique is useful when the problem solution contains repeated or overlapping subproblems
 - Repeated/Overlapping subproblems
 - Meaning if S has subproblems A and B , some smaller subproblems of A are also smaller subproblems of B
 - Choosing an algorithm
 - If you have a simple local criterion to make a choice that leads to an optimal solution - Greedy
 - If not, and subproblems overlap - Dynamic programming
 - If subproblems do not overlap - Divide and conquer
 - Designing a DP program
 - Determine the subproblems we get by making a choice
 - Define a recurrence relation for the **value of the objective function in terms of its values for one of more subproblems**
 - e.g. $\text{opt}(j) = \max(v_j + \text{opt}(p(j)), \text{opt}(j-1))$

- Every DP algorithm uses a table to store values of the objective function for different subproblems
- Determine the "shape" of the table
 - Dimension will depend on the number and range of parameters in the subproblem
 - It's usually an array with as many as dimensions as there are parameters to the subproblem
 - Sometimes we store the subproblem information elsewhere
- Implement the algorithm
 - Approach 1: memoization (recursion)
 - Straightforward recursive approach to compute the value given by the recurrence relation
 - First look at the table to see if this solution is already computed
 - Approach 2: iteration
 - Loop over the problem starting with the base cases
 - Solve later subproblems
 - Make sure that by the time subproblem S needs solution to S' , the solution to S' has already been computed
- Retrieve the optimal solution
 - Start with the original problem and determine the last choice we made
 - Insert this choice into solution
 - Then repeat starting from the subproblem we get after making that choice
- Non-optimization example: Fibonacci numbers
 - $\text{Fib}(0) = 0, \text{Fib}(1) = 1$
 - $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

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

```
# memoized
def fibonacci_mem(n, memo={}):
    if n in memo:
        return memo[n]
    elif n <= 1:
        return n
    # ...
```

- ```
iterative DP version
def fibonacci_iter(n):
 fib = [0, 1]
 for i in range(2, n + 1):
 fib.append(fib[i-1], fib[i-2])
 return fib[n]
```

- Memoization

- Rewrite Brute-Force-Change, to a memoized version

- ```
function memo-change(n):
    create a new array Soln of length
    n
    for i from 1 to n:
        Soln[i] = -1
    return memo-change-helper(n)

function memo-change-helper(i):
    if i < 0:
        return infinity
    else if i = 0:
        return 0
    else if i > 0:
        if Soln[i] == -1:
            Soln[i] = 1 + min(memo-
change-helper(i - 25), memo-change-
helper(i - 10), memo-change-helper(i -
1))
        return Soln[i]
```

- How much time is needed by a call to **memo-change-helper**, not counting the time for recursive calls?
 - $\Theta(1)$, constant time
- Which nodes in the recursion tree do not have children?
 - The nodes that has already computed. The recursive call will terminate at these nodes in repeated calls, and thus will not call further subproblems.
- Give an upper bound on the number of **internal nodes** of the recursion tree on input n
 - Excluding the leaf nodes described from the previous section, there will be at most n nodes in the tree (or $n + 1$ if you count zero).
- Give a big- O upper bound on the number of **leaves** of the recursion tree on input n
 - Only nodes that are internal can make recursive calls, each can make at most 3 calls, thus there are at most $3n$ leaf nodes
- Give a big- O bound on the run-time of algorithm **memo-change**
 - $O(n)$
- Dynamic programming: Growing from the leaves
 - Which entries of the Soln array need to be filled in before we are ready to compute the value for Soln[i]
 - $i - 25, i - 10, i - 1$

- Give a simple order in which we would compute the entries of Soln so that all previous entries needed are **already** computed by the time we want to compute a new entry's value
 - 0 to n
- Rewrite Brute-Force-Change without using recursion

```
function soln'(i):
    # it would be handy if soln had 0
    and negative entries
    # we use this function to simulate
    this
    if i < 0:
        return infinity
    else if i == 0:
        return 0
    else return soln[i]

function DP-Change(n):
    if n <= 0:
        return soln'(n)
    else:
        # assumes n > 0
        create a new array soln[1...n]
        for i from 1 to n:
            soln[i] = 1 + min(soln'(i
- 1), soln'(i - 10), soln'(i - 25))
        return soln[n]
```

- Write an algorithm that uses the values in the Soln array to return the number of coins of each type that are needed to make change with the minimum number of coins

```
function Explain(soln, n):
    if n = 0:
        return []
    if soln[n] == 1 + soln'(n - 25):
        return [25] + Explain(soln, n
- 25)
    else if soln[n] == 1 + soln'(n -
10):
        return [10] + Explain(soln, n
- 10)
    else if soln[n] == 1 + soln'(n -
1):
        return [1] + Explain(soln, n -
1)
```

July 29th - Dynamic Programming in 2D

- Dynamic programming: Growing from the leaves
 - Both **memo-change** and **dp-change** run in the same asymptotic time. Asymptotically in terms of n , how much **memory** do these algorithms use?
 - Both algorithms need $\Theta(n)$, as there are at most n subproblems
 - Imagine you only need the number of coins returned from **brute-force-change**, and don't need to actually calculate change, then for **dp-change** algorithm, how much of the **Soln** array do we need at one time? If we take advantage of this, how much memory does the

algorithm use asymptotically?

- To compute $n = 81$, we only need at least $n = 56$; thus, if to compute $n = 82$, we only need at least $n = 57$.
- For each computation, we only need to keep the 25 most recent elements.
- So, we only need $O(1)$ memory space
- DP in 2D
 - Longest Common Subsequence of two strings: trivial and small instances
 - Trivial: either strings are empty string, both length 1 string
 - Small: "virgil augustus", "agnes victoria"
 - $\text{LCS}(\text{"agnes"}, \text{"augustus"}) = \text{"ags"}$
 - $\text{LCS}(\text{"virgil"}, \text{"victoria"}) = \text{"viri"}$
 - Consider **compute** and **science**. Describe the relationship of the length of their LCS with the length of the LCS of **comput** and **scienc**
 - $\text{LLCS}(\text{"compute"}, \text{"science"}) = 1 + \text{LLCS}(\text{"comput"}, \text{"scienc"})$
 - Consider **tycoon** and **country**. Describe the relationship of the length of their LCS with the length of the LCS of **tycoon** and **countr**
 - $\text{LLCS}(\text{"tycoon"}, \text{"country"}) = \text{LLCS}(\text{"tycoon"}, \text{"countr"})$
 - $\text{"con"} = \text{"con"}$
 - Given two strings **A** and **B** of length $n > 0$ and $m > 0$, we will denote the length of the LCS of **A** and **B** by $\text{LLCS}(\mathbf{A}[1 \dots n], \mathbf{B}[1 \dots m])$. Describe it as a recurrence relation over smaller instances.

$$\text{LLCS}(\mathbf{A}[1 \dots n], \mathbf{B}[1 \dots m]) =$$

$$\bullet \quad \begin{cases} 1 + \text{LLCS}(\mathbf{A}[1 \dots n-1], \mathbf{B}[1 \dots m-1]) & A[n] = B[m] \\ \max(\text{LLCS}(\mathbf{A}[1 \dots n], \mathbf{B}[1 \dots m-1]), \text{LLCS}(\mathbf{A}[1 \dots n-1], \mathbf{B}[1 \dots m])) & \text{otherwise, or} \\ \text{LLCS}(\mathbf{A}[1 \dots n-1], \mathbf{B}[1 \dots m]) & \text{otherwise} \end{cases}$$

- Given two strings **A** and **B**, if either has a length of 0, what is the length of their LCS
- Convert your recurrence into a memoized solution to the **LLCS** problem

```
My solution:
soln[0...n][0...m]
function memo-LLCS(A, B, soln):
    if A == empty or B == empty:
        for each i from 0 to m:
            soln[0][i] = 0
        for each i from 0 to n:
            soln[i][0] = 0
    else if A[n] == B[m]: // Should check if
soln[n][m] is already there
        soln[n][m] = 1 + memo-LLCS(A[1...n-1], B[1...m-1], soln)
    else:
        soln[n][m] = max(memo-LLCS(A[1...n-1], B[1...m], soln), memo-LLCS(A[1...n], B[1...m-1], soln))
    return soln[n][m]
```

- ```

function LLCS(A[1...n], B[1...m]):
 Soln = an (n+1)*(m+1) zero-indexed array
 with entries initialized to -1
 return LLCSHelper(A[1...n], B[1...m],
 Soln)

```

```

function LLCSHelper(A[1...n], B[1...m],
Soln):
 if n == 0 or m == 0:
 return 0
 if Soln[n][m] == -1:
 if A[n] == B[m]:
 Soln[n][m] = 1 +
LLCSHelper(A[1...n-1], B[1...m-1], Soln)
 else:
 Soln[n][m] =
max(LLCSHelper(A[1...n], B[1...m-1], Soln),
LLCSHelper(A[1...n-1], B[1...m], Soln))
 return Soln[n][m]

```

- Extract an actual LCS from an LLCS table

- ```

LCS = ""
function LCS(A[1...n], B[1...m], Soln, LCS):
    if Soln[n][m] == 1 + Soln[n - 1][m - 1]:
        LCS = A[n]/B[m] + LCS
    else if Soln[n][m] == Soln[n - 1][m]:
        LCS = LCS(A[1...n-1], B[1...m], Soln,
LCS)
    else if Soln[n][m] == Soln[n][m - 1]:
        LCS = LCS(A[1...n], B[1...m-1], Soln,
LCS)
    return LCS

```

- ```

function Explain(Soln, n, m):
 if Soln[n][m] == 0:
 return ""
 if A[n] == B[m]:
 return Explain(Soln, n-1, m-1) +
A[n]/B[m]
 else if Soln[n][m] == Soln[n-1][m]:
 return Explain(Soln, n-1, m)
 else if Soln[n][m] == Soln[n][m-1]:
 return Explain(Soln, n, m-1)

```

- Give an iterative solution that produces the same table as the memoized solution

- ```

Soln[0...n][0...m]
function DP-LLCS(A[1...n], B[1...m], Soln):
    for each i from 0 to n:
        for each j from 0 to m:
            if i == 0 or j == 0:
                Soln[i][j] = 0
            else:
                if A[i] == B[j]:
                    Soln[i][j] = 1 + Soln[i - 1][j - 1]
                else:
                    Soln[i][j] = max(Soln[i - 1][j], Soln[i][j - 1])
    return Soln[n][m]

```
- ```

function LLCSiter(A[1...n], B[1...m]):
 Initialize zero-indexed (n+1)*(m+1)
 table: Soln
 Initialize row 0 and column 0 entries to 0
 for i = 1 to n:
 for j = 1 to m:
 if A[i] == B[j]:
 Soln[i][j] = 1 + Soln[i-1][j-1]
 else:
 Soln[i][j] = max(Soln[i-1][j], Soln[i][j-1])
 return Soln[n][m]

```
- Analyze the efficiency of both memoized and iterative in terms of runtime and memory use
  - Memory:  $O(mn)$  - the number of subproblems, for both
  - Runtime
    - DP:  $O(mn)$
    - Memo:  $nm$  subproblems  $\times O(1)$  (non-recursive actions) =  $O(mn)$
- If we only want the length of LCS of **A** and **B** with lengths  $n$  and  $m$ , where  $n \leq m$ , explain how we can "get away" with using only  $O(n)$  memory in the dynamic programming solution.
  - We only need the current row/column and the previous row/column.
  - This should yield either  $O(n)$  or  $O(m)$  depending on how we go through the table

## July 31st - NP-completeness

- NP-completeness
  - Up to now...
    - Every problem has been "easy" to solve - having an algorithm that has  $O(n^k)$  runtime
  - Now
    - We will look at problems that are "hard" to solve
    - We don't know of any efficient algorithms for them
- Decision vs. Optimization problems

- Optimization: we want to find the solution  $s$  that maximizes or minimizes a function  $f(s)$
- Decision: given a parameter  $k$ , we want to know if there exists a solution  $k$  for which  $f(s) \geq k$  or  $f(s) \leq k$
- Two are almost equivalent
- Similar in complexity
  - If we have a solution to an optimization problem, then it gives an answer to the decision problem
  - If we can solve the decision problem efficiently, then we can use binary search on  $k$  to find answer for optimization problem
- so we look at decision problem only
- P and NP
  - P: the set of all problems for which we have an efficient solver
    - Given a problem instance, the solver decides in polynomial time if the answer is Yes or No
  - NP:  $\sim$  we have an efficient certifier
    - If given a solution that returns Yes, it has an efficient certifier to provide proof
  - We have known easy problems (in P), and problems that we *think* are hard (in NP)
  - **Cook's Theorem:** if SAT can be solved in polynomial time, then every problem in NP can be solved in polynomial time
  - A problem that belongs to NP and is as hard as SAT is called **NP-complete**
  - Familiar NP-complete problems
    - SAT, 3-SAT, but **not** 2-SAT
    - Clique
    - Set packing
    - Independent Set
  - Proving a new problem is NP-complete
    - Prove that  $P$  belongs to NP
    - Prove that  $P$  belongs to NP-hard
  - NP-hard: cannot verify solution in polynomial time
- Proving a new problem is NP-complete, step 1
  - We must prove that we can efficiently verify a certificate to the problem
    - Certificate is a "proof" of a Yes answer; a verifier is an algorithm that checks if a certificate is actually a correct proof
    - We can ask a decision problem: Does there exist an  $X$  such that  $Y$  holds
    - Certificate: whatever goes in  $X$  ("valid solution")
    - Verifier: an algorithm that takes an  $X$  and checks whether  $Y$  holds (check whether a "valid solution" is a "good solution")
  - If a certificate can be verified in polynomial time, then  $P$  is in NP
- $\sim$ , step 2

- Prove that  $P$  is **at least as hard as any other problem in NP** (i.e. in NP-hard)
- We prove "at least as hard" via polynomial-time reduction
  - Pick a known NP-complete problem  $P_{NPC}$
  - Give a polynomial-time algorithm that transforms an instance of  $P_{NPC}$  into an instance of  $P$  with the same Yes/No answer
- If you could solve  $P$  efficiently, then you could solve  $P_{NPC}$  as follows
  - Transform it into an instance of  $P$
  - Solve the instance of  $P$  and return the same answer
- Since  $P_{NPC}$  is NP-complete, this means you could solve every problem in NP in polynomial time
- Intuition: since we can use a solver for  $P$  to solve  $P_{NPC}$ , this tells us that  $P$  is at least as hard as  $P_{NPC}$
- How do we solve NP-complete problems
  - Exactly solving an arbitrary instance will take exponential time. Some workarounds
    - Approximation algorithms: have a bound on how bad the solution is
    - Randomization: get a faster running time by allowing the algorithm to fail with some probability
    - Restriction to certain problem types: sometimes certain subtypes of a problem are solvable in polynomial time
    - Heuristics: algorithms that often lack theoretical guarantees but work well enough in practice
- Boolean Satisfiability (SAT)
  - Given an example instance:  
 $(X_1 \vee \bar{X}_2 \vee X_3 \vee X_4) \wedge (X_5) \wedge (\bar{X}_1) \wedge (X_2 \vee \bar{X}_3 \vee \bar{X}_5) \wedge (\bar{X}_2 \vee X_3)$ , is this satisfiable? Give an assignment if possible.
    - $X_1 = F, X_2 = F, X_3 = F, X_4 = T, X_5 = T$
  - For a SAT instance  $I$ , a truth assignment is a potential solution, and the truth assignment is a *good* solution if it satisfies instance  $I$ . Suppose in addition to instance  $I$  you were given a truth assignment represented as an array  $T[1..n]$  where  $T[i]$  is true iff  $X_i$  is true. How long would it take to certify that truth assignment  $T$  is good?
    - $l = \text{length of SAT instance (total number of literals in all clauses)}$ , the certifier is  $O(l)$
    - This demonstrates SAT is in NP
  - A brute force algorithm tries every assignment of truth values, asymptotically, how many truth assignments might this algorithm try?
    - $O(2^n l)$
- 3-SAT and SAT
  - Suppose that  $I$  has a clause with two literals, say  $(\bar{X}_2 \vee X_3)$ . To obtain  $I'$  from  $I$ , we want to replace this clause by one or more clauses, while ensuring  $I$  is satisfiable iff  $I'$  is. How can we do this?
    - Easy, questionably legit:  $(\bar{X}_2 \vee X_3 \vee X_3)$
    - Harder but legit:  $(\bar{X}_2 \vee X_3 \vee Y) \wedge (\bar{X}_2 \vee X_3 \vee \bar{Y})$
  - What if  $I$  has a clause with only one literal, say  $(X_5)$ ?
    - $(X_5 \vee Y \vee Z) \wedge (X_5 \vee Y \vee \bar{Z}) \wedge (X_5 \vee \bar{Y}) \wedge (X_5 \vee \bar{Y} \vee \bar{Z})$



- What about a clause with 4 literals, say  $(X_1 \vee \bar{X}_2 \vee X_3 \vee X_4)$ . What 3-SAT clauses will you replace?
  - $(X_1 \vee \bar{X}_2 \vee Y) \wedge (X_3 \vee X_4 \vee \bar{Y})$ , the addition of  $Y$  ensures half of the literals can satisfy the original clause
- For the construction above, show that if  $I$  is satisfiable then  $I'$  must also be satisfiable
  - $\text{Yes}(l_1 \vee l_2 \vee l_3 \vee l_4) \rightarrow \text{Yes}(l_1 \vee l_2 \vee Y) \wedge (l_3 \vee l_4 \vee \bar{Y})$
  - Suppose we have a truth assignment to  $X$  variables such that  $I$  ( $l_1 \vee l_2 \vee l_3 \vee l_4$ ) is True (has a True literal). At least one literal  $l_1$  to  $l_4$  is True. For  $I'$ , use same  $X$  assignment:
    - If  $l_1 \vee l_2 = T$ , set  $Y$  to  $F$
    - If  $l_3 \vee l_4 = T$ , set  $Y$  to  $T$
  - This leads to Yes to  $I'$
- Show that if  $I'$  is satisfiable, then  $I$  must also be satisfiable
  - $\text{Yes}(l_1 \vee l_2 \vee Y) \wedge (l_3 \vee l_4 \vee \bar{Y}) \rightarrow \text{Yes}(l_1 \vee l_2 \vee l_3 \vee l_4)$
  - If  $Y = T$ , then  $(l_3 \vee l_4) = T$ , meaning one of  $l_3$  or  $l_4$  must be  $T$
  - If  $Y = F$ , then  $(l_1 \vee l_2) = T$ , meaning one of  $l_1$  or  $l_2$  must be  $T$
- Extend to 5-literal clause  $(X_1 \vee \bar{X}_2 \vee X_3 \vee X_4 \vee \bar{X}_5)$ 
  - $(X_1 \vee \bar{X}_2 \vee Y) \wedge (\bar{Y} \vee X_3 \vee X_4 \vee \bar{X}_5)$
  - $(X_1 \vee \bar{X}_2 \vee Y) \wedge (\bar{Y} \vee X_3 \vee Z) \wedge (\bar{Z} \vee X_4 \vee \bar{X}_5)$
- Transform any clause with  $k > 3$  literals  $(l_1 \vee l_2 \vee \dots \vee l_k)$ , How many clauses do you get, What would be the runtime of an algorithm to do this in function of  $k$ 
  - $(l_1 \vee l_2 \vee \dots \vee l_k) \rightarrow (l_1 \vee l_2 \vee X_{i+1}) \wedge (\bar{X}_{i+1} \vee l_3 \vee X_{i+2}) \wedge \dots \wedge (\bar{X}_{i+(k-4)} \vee l_{k-2} \vee X_{i+(k-3)}) \wedge$
- Call the algorithm **Transform-Clause**. Suppose that  $I'$  is obtained from  $I$  by transforming  $(l_1 \vee \dots \vee l_k)$ , explain why  $I$  is satisfiable iff  $I'$  is satisfiable
  - $\text{Yes}(l_1 \vee \dots \vee l_k) \rightarrow \text{Yes}(I')$ 
    - One of  $l_j$  is True for some  $j \in [1, k]$ . Set  $X_{i+(j-1)} = F$  and  $X_{i+(j-2)} = T$ , and extends both ways, every  $X$  that appears after  $l_j$  should be set to  $F$ , and every  $X$  that appears before  $l_j$  should be set to  $T$
  - $\text{Yes}(I') \rightarrow \text{Yes}(l_1 \vee \dots \vee l_k)$ 
    - ???

## Aug 2nd - NP Completeness (Continued)

- 3-SAT and SAT
  - Call the algorithm **Transform-Clause**. Suppose that  $I'$  is obtained from  $I$  by transforming  $(l_1 \vee \dots \vee l_k)$ , explain why  $I$  is satisfiable iff  $I'$  is satisfiable
    - $\text{Yes}(I') \rightarrow \text{Yes}(l_1 \vee \dots \vee l_k)$ 
      - Assume all  $l_i$  takes  $F$ , there is no such assignment for  $X_j$  to satisfy  $I'$ . Therefore, there is at least one  $l_i$  that is assigned to be  $T$ .
      - Since all  $l_i$  takes  $F$ , then  $X_{i+1}$  must be  $T$ , then by this chain,  $X_{i+k-3}$  must be  $T$  as well to satisfy all clauses besides the last, but the last clause will now evaluate to  $F$ , contradicting to  $I'$  being satisfied.
  - Give a reduction from SAT to 3-SAT.
    - Transform instance algorithm

- Length-1 or -2 clauses: Q2.1, 2.2
- Length-3 clauses: leave as it is
- Length-k clauses: Q2.7
- Transform solution algorithm
  - return YES to SAT iff answer to 3-SAT was YES
- Why is the reduction correct?
  - We proved  $SAT \iff 3-SAT$
  - Longer clauses are satisfiable in 3-SAT iff they are satisfiable in SAT, by Q2.8
- What does a reduction tell us?
  - Scenario 1: Reduction's two algorithm take  $O(f(n))$  time and the black box solver for  $B$  also takes  $O(f(n))$  time. What can we say about the running time to solve Problem  $A$ ?
    - $A \leq O(f(n))$
  - Scenario 2: Reduction's two algorithm take  $O(g(n))$  and we know that there is **no algorithm** for problem  $A$  that runs in  $O(g(n))$  time, what do we know about the running time for Problem  $B$ ?
    - There is no algorithm to solve  $B$  in  $O(g(n))$  time
  - Scenario 3: We know if SAT can be solved in polynomial time, then **any** problem in the large set called "NP" can also be solved in polynomial time. What does our reduction from SAT to 3-SAT tell us?
    - If SAT solvable in polynomial time  $\rightarrow P = NP$
    - If 3-SAT solvable in polynomial time  $\rightarrow$  SAT will be solvable in polynomial time
    - If 3-SAT solvable in polynomial time  $\rightarrow P = NP$

## NP-Completeness, or the Futility of Laying Pipe

- Steiner... Something-or-Others
  - Steiner Tree Problem
    - Trivial: no shaded nodes
    - Two nodes
    - One shaded node
    - Two shaded nodes that are neighbours
    - empty instance
    - Two shaded nodes or a "line"
  - Formalize the problem as an *optimization/decision* problem.
    - Does there exist a set of at most  $k$  edges that connect all shaded nodes?
    - $G = (V, E)$ ,  $S \subset V$  shaded nodes
    - potential solution = set of edges
    - good solution = connects all shaded nodes, uses at most  $k$  edges
  - Think about what an optimal solution to ST looks like. Is it a path, cycle, tree, graph? Can we reduce ST to a similar problem we've seen before
    - Tree
    - Similar to minimum spanning tree
  - Prove that the ST problem is in NP.
    - Certificate: does there exist an  $X$  such that  $Y$  holds

- Certificate is a subset of edges
- Verifier: Check if a subset of edges is size at most  $k$  (at most linear) and connect all shaded nodes (BFS or DFS from a shaded node using only edges in the certificate, check that we reach all shaded nodes. Also linear)
- Reduction from 3-SAT to ST
  - Reducing 3-SAT to ST means ST is in NP-hard

## Aug 7th - NP-Completeness, or the Futility of Laying Pipe (Continued)

- Reduction from 3-SAT to ST
  - Give a reduction
    - First consider a node **hub**, shaded
    - For every unique variable in an 3-SAT problem, take  $X$  as an example, construct two paths from **hub** to **pin** (pin for  $X$  and  $\bar{X}$ ) separately, with each path passing through  $X$  and  $\bar{X}$  respectively. That is, **hub**-- $X$ --**pin**, and **hub**-- $\bar{X}$ --**pin**. (This is to guarantee  $X$  and  $\bar{X}$  are not chosen simultaneously). For  $n$  unique variables, we first raise the edge budget  $k$  to be  $2n$ .
    - For every clause, consider it a shaded node. Connect the node with each literal appearance. For example, consider a clause  $(w \vee x \vee \bar{y})$ , then we connect this node to already present nodes  $w$ ,  $x$ , and  $\bar{y}$ . And then we raise the edge budget by 1. Therefore, if there are  $m$  clauses, we raise the edge budget  $k$  to  $2n + m$
  - Analyze the runtime of this reduction
    - For unique variables, the reduction takes  $O(n)$
    - For number of clauses, the reduction takes  $O(m)$
    - For setting  $k$ , the reduction takes  $O(1)$
- Reduction correctness
  - If  $I$  is a Yes-instance of 3-SAT then  $I'$  is a Yes-instance of ST
    - Consider what a solution for 3-SAT looks like and use it to finish the statement: "Since  $I$  is a Yes-instance of 3-SAT, there must be..."
      - a set of truth assignments to  $X_i$  variables such that every clause contains a True literal
    - To prove  $I'$  is a Yes-instance of ST, we similarly try to show the existence of a good solution (working certificate) to that instance, and conclude that therefore the answer to the instance is YES. What does a good solution for ST look like?
      - connects all shaded nodes, uses no more than  $k = 2n + m$  edges.
  - Given a truth assignment that satisfies  $I$ , how would you choose the edges of the good solution of  $I'$ ?
    - Suppose a satisfying assignment for 3-SAT has variable  $X$  set to be False. Which edges should be included in the ST solution? From hub to  $\bar{X}$  and from  $\bar{X}$  to  $p_X$
    - For every variable assigned to True, add edges hub-- $X_i$ -- $p_i$ . For every variable assigned to False, add edges hub-- $\bar{X}_i$ -- $-p_i$ .

- $c_p$  contains at least one True literal, let the first True literal in  $c_p$  be  $l_j$ , then add the edge  $(l_j, c_p)$ .
- Finish the proof to show that the ST certificate actually works
  - For every variable, we include 2 edges, that gives us  $2n$  edges. The hub and all pins are connected.
  - For every clause, since there is a truth assignment, one of the literal in the clause must evaluate to True. Then we select exactly 1 edge that connects this node to a literal node that evaluates to True. This gives us  $m$  edges. All clause nodes are connected.
  - Then we have an ST instance with at most  $k = 2n + m$  edges.
- If  $I'$  is a Yes-instance of ST then  $I$  is a Yes-instance of 3-SAT
  - How many edges must there be in a tree with  $n$  nodes? If you use at most  $k$  edges to form a connected graph, how many nodes can you connect?
    - With  $n$  nodes, there must be  $n - 1$  edges.
    - With  $k$  edges, they can connect at most  $k + 1$  nodes.
  - Show that any solution for the ST instance works.
    - Suppose ST solution connect all shaded nodes using at most  $2n + m$  edges.
    - It can connect at most  $2n + m + 1$  nodes.
    - We need to include all shaded nodes (hub + pins + clause nodes), that is  $1 + n + m$  nodes
    - ST solution can connect at most  $n$  unshaded nodes.
    - Each variable gadget must have exactly 1 unshaded node connected, so that the pins can be connected with the rest of the tree.
    - Construct SAT solution as follows: for variable  $X_i$ , if  $X_i$  is connected to the shaded nodes in ST solution then set  $X_i$  True; if  $X_i$  is not connected in the ST solution, set  $X_i$  to be False.
    - For clause  $c_p$  in SAT,  $c_p$  is a connected shaded node in the ST solution.  $c_p$  is connected to some literal node  $l_j$  that is also connected to the ST solution. This means  $c_p$  contains a True literal  $l_j$  in 3-SAT.  $c_p$  is satisfied by our truth assignment.