Topics on the Final

- Stable Matching
- Greedy Algorithms
- Dynamic Programming
- **Randomized Algorithms (will cover briefly)**
- Divide and Conquer
- Network Flow
- NP-Completeness
- **Computability**
- **Approximation Algorithms**

Covered today

Computability

Key Ideas

- Turing Machines
- Recursive and recursively enumerable sets
- Showing decidability/undecidability

Turing Machine Overview

Informally, a Turing Machine is a computational machine consisting of a tape and input that it reads in, and performs deterministic actions on the tape based on the input.

Turing Machines can be modified to have multiple tapes, read/write heads, etc.

The *language* of a Turing Machine is the set of strings that it accepts in a finite amount of time.

Configurations: A configuration (p, z, n) is a tuple describing all the relevant information about a Turing machine. p represents the current state of the finite control, z is the contents of the tape, and n denotes the position of the read-write head.

If a machine is in the same configuration, it will *always* take the same next step.

Decidability

Let P be some property of a group of strings (inputs to turing machines)

Ex. “Has odd length”, “of the form $a^nb^nc^n$ for some $n \geq 0$”

P is *decidable* := there exists some turing machine which accepts all inputs with property P and rejects all inputs which do not have property P (in a finite amount of time)

(both examples above are decidable)

Sometimes we call a problem decidable if some property defined by that problem is decidable

The Halting Problem

Let P be the property of strings which represent the binary encoding of some turing machine M and some input to that machine X that running M on x halts

If P were decidable, there must exist some K such that for all s = M#x

The Halting Problem

Let P be the property of strings which represent the binary encoding of some turing machine M and some input to that machine X that running M on x halts

If P were decidable, there must exist some K such that for all s = M#x

- K accepts s if M halts on x
- K rejects s if M does not halt on x

Can we let K be a machine which runs M on x and checks if it halts?

The Halting Problem

let P be the property of strings which represent the binary encoding of some turing machine M and some input to that machine x that running M on x halts

If P were decidable, there must exist some K such that for all s = M#x

- K accepts s if M halts on x
- K rejects s if M does not halt on x

Can we let K be a machine which runs M on x and checks if it halts?

- K accepts s if M halts on x ✓
- Does K reject s if M does not half on x? No, it just runs forever without accepting or rejecting

Diagonalization Overview

Used to prove (the property defined in) the Halting Problem is undecidable

- i.e. There exists some K such that for all inputs s = M#x, K accepts s if M halts on x and K rejects s if M does not halt on x

- Idea (proof by contradiction)
  - Suppose there exists some K which decides the halting problem
  - Write out a table which shows every possible s representing some M#x, and whether or not M halts on x (which matches whether or not K accepts s) (Restrict inputs to M to be binary strings)
  - Show that there exists some machine N not shown on the table, with leads to contradiction

	ε	0	1	00	01	10	11	000	001	010	$\cdots$
M_ε	H	L	L	H	H	L	H	L	H	H	
M_0	L	L	H	H	L	H	H	L	L	H	
M_1	L	H	H	L	L	L	H	H	L	H	
M_{00}	L	H	L	H	H	L	H	H	L	L	
M_{01}	H	L	H	L	L	H	L	L	H	H	$\cdots$
M_{10}	H	L	H	H	L	H	H	H	L	H	
M_{11}	L	L	H	L	H	H	L	L	H	H	
M_{000}	H	H	H	L	H	H	H	L	H	L	
M_{001}	L	L	H	L	L	L	L	H	H	L	
M_{010}	H	H	L	L	H	L	L	H	L	L	
$\vdots$					$\vdots$						$\ddots$

Diagonalization

Let K be the machine which decides the halting problem

Let N be a machine which takes in some input x and:

- Constructs machine M_x and writes M_x#x on its tape
- Runs K on input M_x#x
- If K rejects, have N accept x. If K accepts, have N enter an infinite loop

Thus N halts on x iff M_x loops on x

For all Mx, N's behavior and M_x's behavior differ on input x. Thus N ≠ any M_x

This contradicts the fact that every turing machine with binary inputs is represented in the table!

Properties of sets

Recursive:

- A set X is recursive if you can design a *total* Turing Machine M such that L(M) = X.
- M accepts x if x ∈ X, and rejects x otherwise (in finite time)
- Showing that a problem is decidable is the same as showing that a set is recursive.

To show that a set X is recursive, you must design a Turing Machine M with the given properties, and prove that:

- M accepts x in finite time -> x ∈ X
- M rejects x in finite time -> x ∉ X (or the contrapositive)

Properties of sets

Recursively enumerable:

- A set X is recursively enumerable if you can design a Turing Machine M such that L(M) = X.
- M accepts x in a finite amount of time if $x \in X$, and either rejects x or runs infinitely otherwise.

- HP is recursively enumerable.

To show that a set X is recursively enumerable, you must design a Turing Machine M with the given properties, and prove that:

- M accepts x in finite time -> $x \in X$
- $x \in X$ -> M accepts x in finite time

To show that a set is recursive/r.e., we construct a Turing Machine M that has the properties given above. To show that a set is not recursive/r.e., we use reductions.

Reductions

Now that we've shown the undecidability of one problem, we can use reductions to prove that other problems are undecidable.

To show that a problem Z is undecidable, we need to show that if could solve Z, we would be able to solve the Halting Problem for any machine M and input x.

Let HP be the set representing the Halting Problem: $\{M\#x \mid M \text{ halts on } x\}$

Let B be the set representing problem Z

We would like to show that HP $\leq_m$ B. Formally, this involves constructing a mapping σ that maps all elements of HP to some element in B, and all elements not in HP to some element not in B. σ must be *total* and *effectively computable*.

Properties

Theorem 2.

(i) *If* $A \leq_m B$ *and* B *is r.e., then so is* A*. Equivalently, if* $A \leq_m B$ *and* A *is not r.e., then neither is* B*.*

(ii) *If* $A \leq_m B$ *and* B *is recursive, then so is* A*. Equivalently, if* $A \leq_m B$ *and* A *is not recursive, then neither is* B*.*

By 2 (ii), we can reduce from HP or ~HP to B to show that B is not recursive (and hence the problem that B represents is undecidable)

By 2 (i), we can reduce from ~HP to B to show that B is not r.e.

You can also use any suitable problem from course material that is not recursive/r.e. in your reductions.

Steps to showing undecidability

Let the unknown problem **Z** be represented by the set B, consisting of yes-instances of the problem.

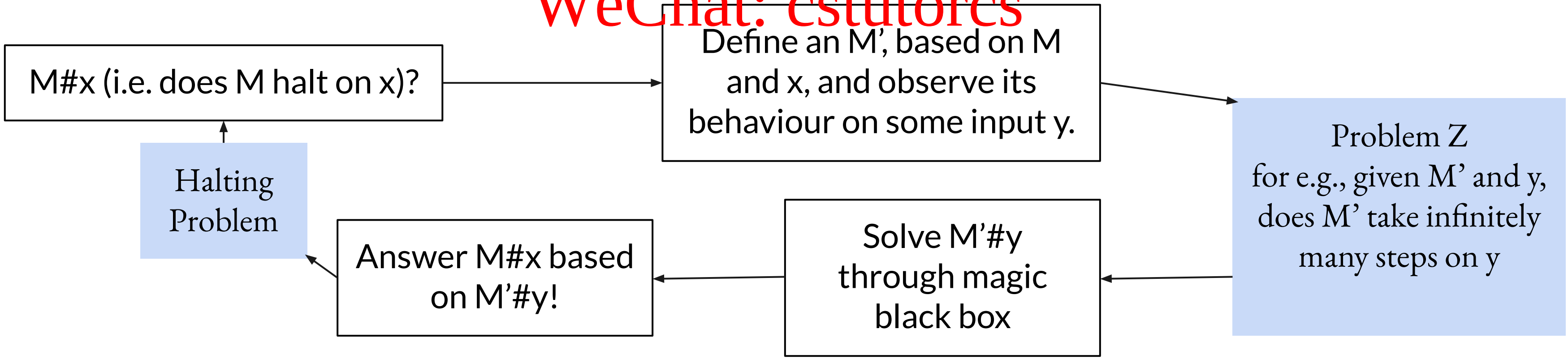
Pick a known undecidable problem to reduce from. This will almost certainly be the Halting Problem (or the co-Halting Problem).

1. Take an arbitrary instance of the Halting Problem - a machine M and an input *x*.
2. Formally, we want to map to an instance of the unknown problem (i.e. an element of B).
    - Construct a machine M' and describe its behaviour on some input *y*.
    - Want to construct M' in a way that we could use problem Z to infer a property of M', and hence conclude whether M halts or does not.
3. Prove your reduction. Either:
    - $M\#x \in HP \Leftrightarrow M'\#y \in B$, or
    - $M\#x \notin HP \Leftrightarrow M'\#y \in B$ (this is the same as showing $\sim HP \leq_m B$).
4. The reduction implies that if Z were decidable, HP would be decidable, which isn't possible - hence Z is undecidable.

Note: the problem instances might not necessarily be of form M'#y. For e.g., in the HW problem about deciding whether 2 machines agree on an input, an instance of B would consist of 2 machines and an input *y*.

Tips

- When asked if a problem is decidable or undecidable, first take a minute to see if you can construct a total Turing Machine to solve the problem. If you run into a familiar difficulty, for e.g. your machine would loop forever on some cases, switch to thinking of reductions!
- The hardest part of a reduction is coming up with the construction M’ - the proof will usually just involve writing out how your machine will behave, by definition, when given a yes or no-instance of the Halting Problem.



Example

3. Determine whether the following decision problem is decidable or not. Explain why your answer is correct.

   Given a Turing Machine M and two input strings x and y. Does M output the same result for x and y? That is, is it the case that, if M accepts, rejects or doesn't halt on x, does it do the same for y?

Decidable or undecidable?

Let the problem be Z. First, make clear what counts as an instance of Z - in this case, it is some machine M', and 2 input strings *y* and *z*.

Hence, given some M and x for the Halting Problem, we want to construct an M' and observe its behaviour on 2 input strings. **(we get to decide M', and the 2 input strings)**

Example

Let HP = {M#x | M halts on x} (Represents the halting problem)

Let B = {M’, y, z | M’ behaves the same on both inputs y and z }

We want to show that the Halting Problem can be solved if we can solve Z, by constructing a reduction.

Let us take an arbitrary instance of HP - some turing machine M and input x.

We would like to construct a Universal Turing Machine M’ and inputs y, z, such that:

(M’ behaves the same on y and z) iff (M **does not halt** on x)

(This is our mapping)

Example

Step 1: Define the mapping.

We create an instance M', y, z. Let y be ε, and z be any other input string. On ε, let M' loop forever. On any other input z, let M' have the following behaviour:

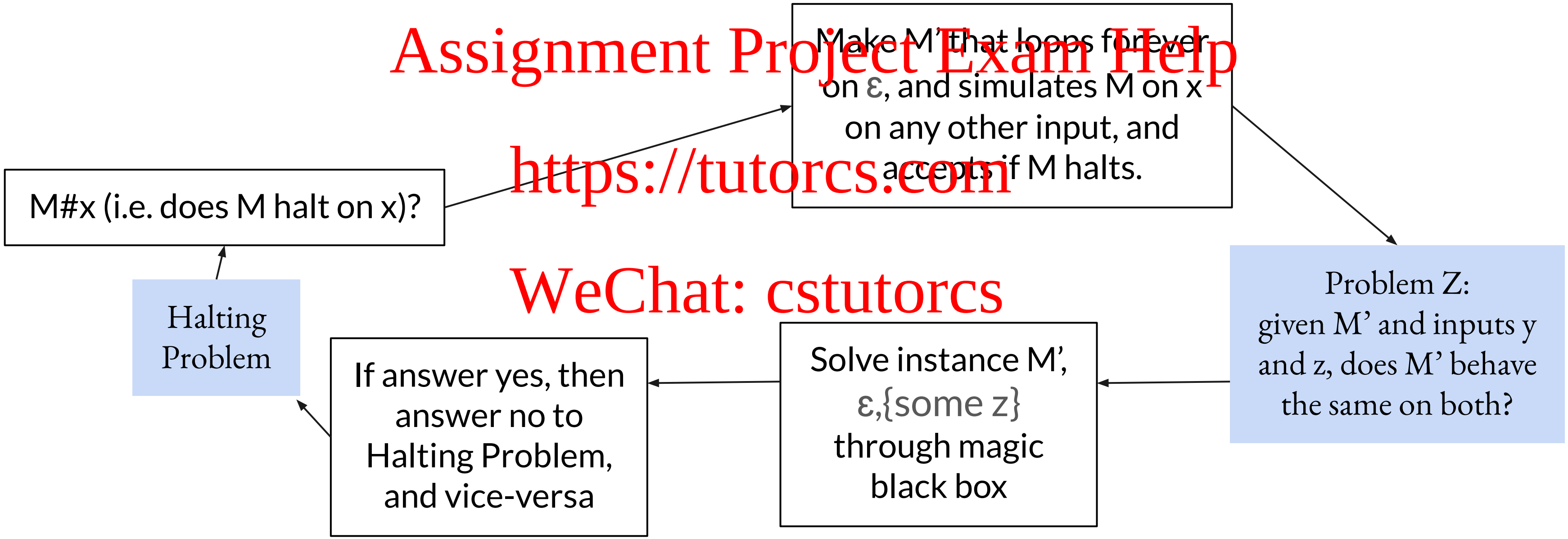
1. Erase z from the tape, and start simulating M on x.
2. If M halts, M' halts and accepts (hence, otherwise loops forever).

Step 2: Show that if M does not halt on x, then M' behaves the same on ε and z.

Step 3: Show that if M halts on x, then M' behaves differently on ε and z.

Proof

Example


M#x (i.e. does M halt on x)?
Make M' that loops forever on ε, and simulates M on x on any other input, and accepts if M halts.
Problem Z: given M' and inputs y and z, does M' behave the same on both?
Solve instance M', ε,{some z} through magic black box
If answer yes, then answer no to Halting Problem, and vice-versa
Halting Problem

Approximation algorithms

α-approximation

- If maximization:
  - $\alpha \cdot$ APX ≥ OPT or APX ≥ OPT/α
  - If α = 2, want to show our approximate algorithm solution is at least half optimal
- If minimization
  - APX/α ≤ OPT or APX ≤ $\alpha \cdot$ OPT
  - If α = 2, want to show our approximate algorithm solution is at most twice the optimal
- Examples in which the approximation algorithm returns a solution that is worse than optimal (or, is α off from optimal)

Important concepts/algorithms

- Greedy approximation algorithms
  - Vertex Cover
  - Set Cover
  - Knapsack
  - Arbitrarily good Knapsack Approximation

Minimum Vertex Cover

- Given undirected graph G = (V, E), find a vertex set C ⊆ V that contains an endpoint of each edge and has **as few vertices as possible.**

- A greedy algorithm gives a 2-approximation
  - Since this is a minimization problem, the vertex cover returned by the approximation algorithm has size at most twice the optimal

Approximation Algorithm for Vertex Cover

- Algorithm:
    - Start with VC’ = ∅
    - While there exists an edge (u, v), add u and v to VC’, and delete every edge having u or v as an endpoint
    - Output VC’

Showing that Greedy is a 2-Approximation

- After k iterations of the while loop, we've considered k edges that are not adjacent
  - And the greedy vertex cover has size |VC'| = 2k
- The optimal vertex cover VC* must contain an endpoint for each of the k non-adjacent edges chosen in the while loop, so |VC*| ≥ k.
- So 2 * |VC*| ≥ 2k = |VC'|

Tight Example for Greedy Vertex Cover

- Greedy always takes both vertices, but the optimal vertex cover is just one of them
- So the greedy solution is twice the size of optimal

Set Cover (Section 11.3)

- Given set U of n elements and a list $S_1, \ldots, S_m$ of subsets of U with corresponding weights $w_1, \ldots, w_n$, a set cover I, is a collection of these sets whose union is equal to all of U
- Goal: Find a set cover that minimizes the total weight, $\Sigma_{S_i \in I} w_i$
- We will show a greedy approximation algorithm for Set Cover

Approximation Algorithm for Set Cover

- Greedy Rule: Choose the set S_i that minimizes the "cost per new element covered" $w_i/|S_i \cap R|$, where R is the set of remaining uncovered elements
- Greedy-Set-Cover:
  - $R = U$, $I = \emptyset$
  - While $R \neq \emptyset$:
    - Select S_i that minimizes $w_i/|S_i \cap R|$
    - Delete S_i from R
  - Return I

Approximation Factor for Greedy Set Cover

- This algorithm is an **H(d)** approximation algorithm.
  - $d = \max_i |S_i|$
  - $H(d) = \Sigma_{i \in \{1,\ldots,d\}} (1/i)$ and is the d^{th} Harmonic number
- See lecture notes and section 11.3 for a proof of this

Knapsack Problem

- Given *n* items each with weight w_i and value v_i, and a max allowed weight *W*, select a subset *I* of items of maximum total value $\sum_{i \in I} v_i$ whose total weight $\sum_{i \in I} w_i \leq W$.
- A greedy algorithm gives a 2-approximation
  - Since this is a **maximization** problem, the approximation algorithm solution has value at least **half** of optimal

2-Approximation Algorithm for Knapsack

- Use 2 greedy algorithms
  - Greedy A: Sort items in order of their value. Add to set *I* till weight is not exceeded.
  - Greedy B: Sort items in order of their density. Add to set *I* till weight is not exceeded.
- Run both Greedy A and Greedy B, and take the better of the two. This yields a total value at least ½ optimal.

Example Where Greedy A Does Badly

- 1 item of weight W and value W, and (n-1) items of weight 1 and value W-1
- Then Greedy A takes the 1 item of weight W and value W
- But the optimal would be to take the (n-1) items, with a cumulative weight of (n-1)(W-1)

- However, Greedy B will choose the optimal here

Example Where Greedy B Does Badly

- Item 1 has weight 1 and value $1+\varepsilon$, and item 2 has weight W and value W

- Clearly item 1 has higher density, so Greedy B will take item 1 for a value of $1+\varepsilon$
- But the optimal is to take item 2 for a value of W

- However, Greedy A will take the optimal here

Showing that Greedy Knapsack is a 2-Approximation

- Let I be the solution obtained by Greedy B, and let j be the first item that Greedy B can't include. Then $v_j + \sum_{i \in I} v_i \geq OPT$

- Additionally, Greedy A returns a solution that includes the item with the maximum value, or $\max_i\{v_i\}$

- $OPT \leq v_j + \sum_{i \in I} v_i \leq$ Greedy A + Greedy B $\leq 2 * \max\{$Greedy A, Greedy B$\}$

Tight Example(s) for 2-Approx. Greedy Knapsack

$W = 2$

Item 1	$w_1 = \varepsilon$	$v_1 = 2\varepsilon$
Item 2	$w_2 = 1$	$v_2 = 1 - \varepsilon$
Item 3	$w_3 = 1 + 2\varepsilon$	$v_3 = 1$
Item 4	$w_4 = 1 - 2\varepsilon$	$v_4 = 1 - 4\varepsilon$

- Items are ordered by density
  - Greedy B chooses items 1 and 2 -- value $1 + \varepsilon$
  - Greedy A chooses item 3 -- value 1
  - So the greedy approximation takes Greedy B's approach
- But optimal chooses items 3 and 4 -- value $2 - 4\varepsilon$
- As we send $\varepsilon \to 0$, the ratio between opt and greedy values goes to 2

Arbitrarily Good Knapsack Approximation

- Algorithm:
  - Eliminate all items with weight greater than W and choose parameter **ε**
  - Set $b = (\varepsilon/(2n)) * \max_i v_i$; set new values $v_i^* = \lceil v_i / b \rceil$
    - Then $v_i^* \leq \lceil 2n/\varepsilon \rceil$
  - Run Knapsack DP (which runs in pseudopolynomial time) for items with original weights and v_i^* values
- Runtime: $O(n^3 / \varepsilon)$
- Approximation factor: $(1 + \varepsilon)$ for any $\varepsilon > 0$
- Idea for why this works: small changes to the input values shouldn't change the optimal solution by much

Greedy Approximation Practice (K&T chapter 11 problem 1)

Summary: You have *n* containers of weight w1, w2, ...wn. You also have a set of trucks. Each truck can hold *K* units of weight. You can only load one truck at a time. The goal is to minimize the number of trucks needed to carry all the containers(this is different than the other truck loading problem in the greedy section as you have access to all the items to load before loading so no constraint on order).

A greedy algorithm one might use is to simply start with an empty truck and pile containers 1, 2, 3... into it until you can't fit another container. Send this truck off and do the same for a fresh truck. This algorithm, which considers trucks one at a time, may not achieve the most efficient packing.

1. Give an example of a set of weights and a value of *K* where this greedy algorithm does not use the minimum possible number of trucks
2. Prove that this greedy algorithm is a 2-approximation of the optimal solution. That is, it will never use more than 2 times the optimal number of trucks needed.

Solution part 1

One example is with weights 2, 3, 2, 1 and K=4.

Greedy will use 3 trucks and separate items as {2}, {3}, {2,1}

However the optimal is 2 trucks: {2,2}, {3,1}

Solution part 2

1. First, lower bound the value of OPT:
   a. Let $W = \Sigma w_i$.
   b. We know that the optimal must use at least $\lceil W/K \rceil$ trucks: This is as using exactly W/K trucks means that each truck is packed fully which is the absolute best case scenario

2. Next, upper bound how our greedy algorithm does:
   a. Recall that our greedy algorithm always packs items onto trucks if possible.
   b. This means that for any pair of trucks, the combined load of the two trucks $>K$. If not, this means the load on both trucks could be fit onto 1 truck which our greedy algorithm would have done.
   c. Now, the final step is to consider even number of trucks and odd number of trucks.

Solution part 2 (cont) - even trucks

- On average, every truck must have load $> K/2$. (Prove for yourself)
- The total number of trucks used by our algorithm $< W/(K/2) = 2W/K$ as $W/(K/2)$ corresponds to each truck being half full.
- We have $2W/K > ALG >= OPT >= W/K$. If we rewrite this, we get $ALG/OPT < (2W/K)/(W/K) = 2$ which means $ALG < 2*OPT$ which is a 2-approximation.

Solution part 2 (cont) - odd trucks

- Let the number of trucks ALG = 2n + 1. The first 2n trucks have combined weight > nK (think about why!)
- This means that OPT must use at least n+1 trucks. n trucks is not enough as in the best case, this would only hold nK total weight but we know that the combined weight of all our items > nK.
- We have OPT >= n+1 and ALG = 2n+1. This means ALG/OPT = (2n+1)/(n+1) <= (2n+2)/(n+1) = 2. Thus, in the case of odd trucks we have also shown that the greedy algorithm is a 2-approximation.

Randomized algorithms

Algorithms/Key Ideas We've Covered

- Median Finding (Divide and Conquer)
- Closest Pair of Point

Key Idea

- Use randomization within algorithm to solve problems more efficiently than deterministic ones

- Typically evaluated through "Average Case" analysis (rather than "Worst Case" analysis) and probabilities of certain behavior (ex. correct/incorrect)

- Reasoning about probabilities also required to analyze performance

Probability Concepts to Know

(Probability recitation slides go into a lot more detail)

- If A and B are independent, P(A and B) = P(A) * P(B)
- The expected value of a random variable X can be thought of as a weighted mean of the different values X can take.

- If X can be written as the sum of random variables Y_1 + Y_2 + …, E(X) = E(Y_1) + E(Y_2) + …
  - Does not require Y_1, Y_2, … to be independent!

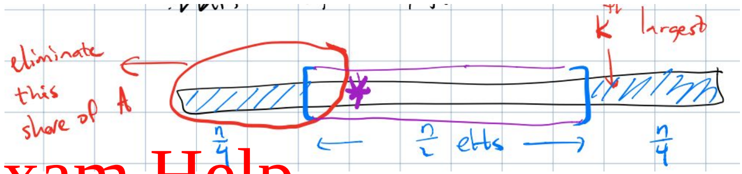
- If you can model a random variable as a distribution covered in class, you can use the E(X) formula covered in class
  - X ~ Geom(p): E(X) = 1/p
    - Number of coin flips needed to get a heads (p = ½)
  - X ~ Binom(n, p): E(X) = np
    - Number of heads in 5 coin flips (n = 5, p = ½)

Median Finding

- Goal: Find the k'th largest element in a list L of n numbers
- Idea 1:

  - Choose a pivot x in L (ex. First element)
  - Partition L into elements greater than x (L^+)and less than x (L^-) - Takes (O(n) time)
  - Recurse on whichever of L^+ and L^- contains the median (determined by looking at sizes of L^+, L^-, and previous partitions)

  - Worst case $O(n^2)$ time: could choose min element as splitter each time, and next set size decreases by 1



Median Finding

- Goal: Find the k'th largest element in a list L of n numbers
- Idea 2 (Randomization): Randomly choose pivot, only recurse once we've found a "good" pivot
    - Randomly choose an pivot x in L
    - Partition L into elements greater than x (L^+)and less than x (L^-)
    - If the chosen pivot is not contained in the middle 50% of the list, go back to step 1
    - Recurse on whichever of L^+ and L^- contains the median (determined by looking at sizes of L^+, L^-, and previous partitions)
    - Theoretically could run forever
    - In expected case
        - Probability of pivot falling is ½, so the expected number of pivots needed per recursive step is 2
        - At most ¾ of the list is passed to the recursive call
        - Recurrence is $T(n) \leq T(3n/4) + 2n$, so runtime is $O(n)$ by Master Theorem

Closest Pair Of Points

Previously solved using Divide and Conquer

Idea: Randomly sort the points and then put them in a dictionary (using hashing). Otherwise most distance/lookups operations are the same to compare with adjacent cells. Use hashing to bring lookup and insertion time to O(1) amortized

Practice Problem (Chapter 13, Problem 8, simplified)

For a graph G = (V, E), let an induced subgraph of G be one which contains some subset of vertices X and all edges with both endpoints in X

Describe a randomized algorithm which when given a graph G and number k ≤ |V|

- Finds some subset of k vertices that form an induced subgraph with high expected density
  - i.e. Expected number of edges in the induced subgraph is ≥ |E| • (k choose 2)/(|V| choose 2) edges)
- Runs in O(|V|) time

Practice Problem, Solution

- Solution: The algorithm just chooses k vertices at random and returns them
- Why does this work?
  - Probability that a single randomly sampled pair of vertices are connected by an edge in G?

Practice Problem, Solution

- Solution: The algorithm just chooses k vertices at random and returns them
- Why does this work?
  - Probability that a single randomly sampled pair of vertices are connected by an edge in G
    - There are m edges in the graph, and n choose 2 pairs of vertices → |E|/(|V| choose 2)
  - Expected number of edges in induced subgraph of k randomly chosen vertices?

Practice Problem, Solution

- Solution: The algorithm just chooses k vertices at random and returns them
- Why does this work?

  - Probability that a single randomly sampled pair of vertices are connected by an edge in G
    - There are m edges in the graph, and n choose 2 pairs of vertices → |E|/(|V| choose 2)
  - Expected number of edges in induced subgraph of k randomly chosen vertices:
    - For any pair of vertices n_i and n_j among the k chosen vertices, let X_{ij} be an indicator random variable which is 1 if there's an edge between n_i and n_j and 0 otherwise
    - $E[X_{ij}]$ = Probability of an edge between a randomly chosen pair = |E|/(|V| choose 2)
    - Total Number of Edges = ΣX_{ij}
    - E[Total Number of Edges] = $E[\Sigma X_{ij}] = \Sigma E[X_{ij}]$ = (k choose 2) • $E[X_{ij}]$ = |E| • (k choose 2)/(|V| choose 2)