CS 125 **Final Review** December 8, 2016

Contents

1 Notes about the Final

- The final is **Wednesday, December 14, 2016 at 2pm in Emerson Hall 101**.
- We will have a review section to go over these materials on **Thursday, December 8 from 5-7pm in MD221**. We won't go through all of this material during the review section, so bring questions/problems/concepts that you'd like to talk about.
- These review notes are not comprehensive. Material not appearing here is still fair game.
- Some study tips:
  - For any algorithms we covered in class, you should know what it does, the runtime, and and amount of memory required (if applicable).
  - For models of computation that we've talked about, you should know the definition, useful variants, and their relative power.
  - For complexity classes covered, you should also know techniques for determining whether a language is in a class. You should also know some examples for any given class.
  - Know how to apply any of the techniques we've covered. Apart from the problems here, we will post links on Piazza towards sources of practice problems.

2 Mathematics

2.1 Big-O and the Master Theorem

Big-O notation is a way to describe the rate of growth of functions. In CS, we use it to describe properties of algorithms (number of steps to compute or amount of memory required) as the size of the inputs to the algorithm increase.

$f(n)$ is $O(g(n))$	if there exist c, N such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.	f "$\leq$" g
$f(n)$ is $o(g(n))$	if $\lim_{n\to\infty} f(n)/g(n) = 0$.	f "$<$" g
$f(n)$ is $\Theta(g(n))$	if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$.	f "$=$" g
$f(n)$ is $\Omega(g(n))$	if there exist c, N such that $f(n) \geq c \cdot g(n)$ for all $n \geq N$.	f "$\geq$" g
$f(n)$ is $\omega(g(n))$	if $\lim_{n\to\infty} g(n)/f(n) = 0$.	f "$>$" g

A very useful tool when trying to find the asymptotic rate of growth (Θ) of functions given by recurrences is the Master Theorem. The solution to the recurrence relation $T(n) = aT(n/b) + cn^k$, where $a \geq 1$, $b \geq 2$ are integers, and c and k are positive constants, satisfies

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

Exercise. *Use the Master Theorem to solve* $T(n) = 3T(n/2) + 19\sqrt{n}$

Solution.
Using the Master Theorem, we find $3 > 2^{0.5}$, so $\Theta(n^{\log_2 3})$

Exercise. *Using Big-Oh notation, describe the rate of growth of* $\log n$ *vs.* $n \max\{n - 2\lfloor n/2 \rfloor, 0\}$.

Solution.
The two functions are incomparable, because $\max\{n - 2\lfloor n/2 \rfloor, 0\}$ oscillates between 0 and 1 as n changes parity, so the second function oscillates between n and 0.

2.2 Countability

A set S is **countable** (either finite or countably infinite) if and only if there exists a surjective mapping $\mathbb{N} \to S$, i.e. if and only if we can enumerate the elements of S. This means that the following are countable:

- Any subset of a countable set.
- A countable union of countable sets.
- The direct product of two countable sets.
- The image of a countable set under a set function.

Exercise. *True or false:*

1. *The number of languages* not *in P is countable.*
2. *The set of* finite *languages over alphabet* $\{a, b\}$ *is uncountable.*

Solution.
We have:

1. This is false. For each language in P, we can pick a Turing machine that decides it in polynomial time, and different languages will get different Turing machines. Since the set of all Turing machines is countable, the class P is also countable. However, the set of all languages is uncountable - so, the set of languages outside P is uncountable.
2. This is false. Each finite language over $\{a, b\}$ can be represented as a finite string over some alphabet Σ (where Σ is $\{a, b\}$ plus some additional characters). But the set of all finite strings over Σ is countable.

2.3 Probability

- A discrete random variable X which could take the values in some set S can be described by the probabilities that it is equal to any particular $s \in S$ (which we write as $\mathbb{P}(X = s)$).
- Its expected value $\mathbb{E}(X)$ is the "average" value it takes on, which is equal to $\sum_{s \in S} s \cdot \mathbb{P}(X = s)$.
- If some event happens with probability p, the expected number of independent tries we need to make in order to get that event to occur is $1/p$.
- $\displaystyle\sum_{i=0}^{\infty} p^i = \frac{1}{1-p}$ (for $|p| < 1$) and $\displaystyle\sum_{i=0}^{n} p^i = \frac{1-p^{n+1}}{1-p}$ (for all p).
- For a random variable X taking on nonnegative integer values, $\mathbb{E}(X) = \displaystyle\sum_{k=0}^{\infty} \mathbb{P}(X > k)$.
- *Linearity of expectation*: for any random variables X, Y, $\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$. In particular, this holds even if X and Y are not independent (for example, if $X = Y$).
- *Markov's inequality*: for any nonnegative random variable X and $\lambda > 0$, $\mathbb{P}(X > \lambda \cdot \mathbb{E}(X)) < \frac{1}{\lambda}$. In other words, this indicates that the probability of X being significantly larger than its expectation is small.

Exercise. *Suppose you have a fair die with n faces. Find an upper bound for the probability that after m rolls, you have not rolled all n possible values. How many times should you roll the die if you want a failure probability of at most P of seeing all n possible values?*

Solution.
The probability that we never see a given value is $\left(\frac{n-1}{n}\right)^m$. Therefore, by union bound, the probability that there is at least one value we never see is at most $\left(\frac{n-1}{n}\right)^m \cdot n \leq ne^{-m/n}$.

For the second part, we directly apply the result from the Coupon Collector Problem (from sectio n notes) to get that it takes $\Theta(n \log n \log \frac{1}{P})$ rolls.

Exercise. *The exam is over, and the TFs and professors are entering the final scores into our spreadsheet. As we go through the pile, we like to keep track of the highest score.*

Let us model the problem as follows: assume test scores are distinct (no ties), there are n people in the class, and the pile of exams is in a completely random order when we start entering scores.

1. *What is the probability the ith exam we enter is a new high score when we see it, for $i = 1, \ldots n$.*
2. *What is the expected number of high scores we see as a function of n?*

Solution.
We have:

1. If we consider the tests in the first i positions, then each of those tests is in the ith position with probability $1/i$. Hence, the probability that the largest of those tests is in the ith position is also $1/i$.

   Alternatively, an equivalent way to think about this problem is that we draw a uniformly random permutation π of the numbers $\{1, 2, \ldots, n\}$ and these represent the ordering of the scores. Now the question is to find

   $$\mathbb{P}(\forall j < i : \pi(i) > \pi(j))$$

   Observe that this probability is the number of permutations σ in which $\forall j < i : \sigma(i) > \sigma(j)$, divided by the number of all permutations, $n!$. We can count all the σ with this property like that: the last $n - i$ positions of σ can be anything, in any order, for which there are $n \times (n-1) \times \ldots \times (i+1)$ possibilities. Given those, $\sigma(i)$ has to be the largest of the remaining numbers, and then the first $i - 1$ positions of σ are the remaining numbers after that, in any order, for which there are $(i-1) \times \ldots \times 2 \times 1$ possibilities. Thus there are $n \times (n-1) \times \ldots \times (i+1) \times (i-1) \times \ldots \times 2 \times 1 = n!/i$ possiblities for σ, which gives $\mathbb{P}(\forall j < i : \pi(i) > \pi(j)) = 1/i$.

2. The number of high scores is

   $$X = X_1 + X_2 + \ldots + X_n$$

   where X_i is the indicator random variable of the event $\{\forall j < i : \pi(i) > \pi(j)\}$. Hence,

   $$\mathbb{E}(X) = \sum_{i=1}^{n} \mathbb{E}(X_i) = \sum_{i=1}^{n} 1/i = \Theta(\log n)$$

   (and in fact tighter bounds can be given to show that the latter sum is very close to $\log n$).

3 Models of Computation

3.1 DFAs & NFAs

A **deterministic finite automaton** (DFA) M is a 5-Tuple $(Q, \Sigma, \delta, q_0, F)$:

- Q: a finite set of states.
- Σ: the input alphabet.
- δ: a transition function $Q \times \Sigma \to Q$. If $\delta(p, \sigma) = q$, then if M is in state p and reads symbol $\sigma \in \Sigma$ then M enters state q (while moving to next input symbol).
- q_0: a start state in Q.
- F: the accept or final states (a subset of Q).

We showed in class that a DFA can perform bounded counting and pattern recognition.

A **nondeterministic finite automaton** (NFA) N is a 5-Tuple $(Q, \Sigma, \delta, q_0, F)$ with the following difference from a DFA:

- δ: a transition function $Q \times (\Sigma \cup \{\epsilon\}) \to P(Q)$. If $\delta(p, \sigma) = Q'$, then if N is in state p and reads symbol $\sigma \in \Sigma$ then N can enter any state $q \in Q'$ (while moving to next input symbol). If $\sigma = \epsilon$, then N can jump to any state in Q' without moving the input head.

Remember that $\emptyset \in P(Q)$ as well, so the transition function may not send us anywhere under some symbols (which is indicated by a lack of an arrow labeled by that symbol in the state diagram).

It is often easier to explicitly construct an NFA for a language than to construct a DFA for it. However, using the **subset construction**, we know that any NFA has an equivalent DFA from the point of view of computability (but possibly with exponentially many states).

Exercise. *Draw an NFA for the language (with alphabet $\Sigma = \{a, b\}$)*

$$L = \{s : s \textit{ begins and ends with the same letter}\}.$$

Write down a formal description of your NFA.

Solution.

Exercise. *Using the subset construction, draw a DFA for the previous language.*

Solution.

Exercise. *True or false: it is possible to determine algorithmically whether a DFA accepts only finitely many strings.*

Solution.
True. Observe that a DFA accepts an infinite language iff there is a cycle in its state diagram that is reachable from the start state, and from which a finite state is reachable - indeed, if there is such a cycle, we can use it to make the DFA accept arbitrarily long strings, and if there is no such cycle, the DFA can only accept strings of length $\leq |Q|$.

For the algorithm, represent the state diagram as a directed graph, and ignore all vertices that are either not reachable from the start state, of from which there is no path to any final state. Then search for cycles in the remaining graph. This can be done by DFS in linear time.

3.2 Turing Machines

A **(deterministic) Turing Machine** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{halt})$:

- Q is a finite set of states, containing a start state q_0, and a halt state q_{halt}.
- Σ is the input alphabet.
- Γ is the tape alphabet, containing Σ and $\sqcup \in \Gamma - \Sigma$ (the blank symbol).
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function. If $\delta(p, \sigma) = (q, \sigma', d)$, then if M is in state p and reads symbol $\sigma \in \Sigma$ then M enters state q, overwrites σ with σ', and moves the tape head one unit in direction d.

A **non-deterministic Turing Machine** (NTM) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{halt})$ with the following difference from a TM:

- δ: a transition function from $Q \times \Gamma$ to $P(Q \times \Gamma \times \{L, R\})$, so there can be several options for the next step at each stage, and the NTM can follow any of them.

We showed in class that there exists a **universal TM** – i.e. one which given a pair $\langle M, w\rangle$ consisting of a TM M and a string w, simulates w on M (with only polynomial slowdown). Similarly, we can build a nondeterministic universal TM for a pair $\langle N, w\rangle$ (with only polynomial slowdown) or a deterministic universal TM for a pair $\langle N, w\rangle$ (with possibly exponential slowdown).

Exercise. *Define a* Stay-Still TM (SSTM) *to be like an ordinary TM, but with an additional possibility of staying still in a transition (instead of being required to move left or right in each step).*

Show, using implementation-level descriptions, that SSTMs are equivalent in power to TMs.

Solution.
We have to show that an SSTM can simulate a TM, and vice versa. The first statement is obvious, as a SSTM is allowed to do strictly more things than an ordinary TM.

For the other direction, to simulate an SSTM by a TM, take the state diagram of the SSTM, and replace all instances of a transition $q \to q'$ by standing still and writing some symbol by first entering a new intermediate state q'', going right and writing the same symbol via $q \to q''$, and then going left without changing anything via $q'' \to q'$. This removes all 'stay still' commands, while preserving the final results of the computation.

3.3 Word-RAMs

A **word-RAM program** is any finite sequence of instructions $P = (P_1, P_2, ..., P_q)$. Valid instructions are integer arithmetic operations, bitwise operations, saving to or loading from memory, conditional GOTOs, HALT, and MALLOC.

A configuration of a word-RAM program P is a tuple $C = (l, S, w, R, M)$git

- $l \in \{1, \ldots, q + 1\}$ is the program line counter.
- $S \in \mathbb{N}$ is the space usage.
- $w \in \mathbb{N}$ is the word size.

- $R = (R[0], ..., R[r-1]) \in \{0, \ldots, 2^{w-1}\}^r$ is the register array.
- $M = (M[0], ..., M[S-1]) \in \{0, ..., 2^{w-1}\}^S$ is the memory array.

We showed in class that word-RAMs and TMs are equivalent up to polynomial slowdown.

4 Complexity

4.1 Regular Languages

A language L is **regular** if there is a DFA (equivalently, an NFA) that accepts it. We know how to determine whether a language is regular:

Myhill-Nerode: A language $L \subset \Sigma^*$ is regular if and only if there are only finitely many equivalence classes under the following relation $\sim_L$ on Σ^*:

$$x \sim_L y \iff \forall z \in \Sigma^* : xz \in L \iff yz \in L.$$

The minimum number of states in a DFA for L is exactly the number of such equivalence classes.

As we saw in section, the class of regular languages is closed under union, intersection, complement, concatenation, Kleene star, homomorphisms and inverse homomorphisms.

Exercise. *Let $L = \{wa^n : w \in \{a,b\}^*, n =$ the number of a's in $w\}$. Is L regular?*

Solution.
No. For any $j > i$, a^i and a^j are in different equivalence classes because $a^iba^i \in L$ but $a^jba^i \notin L$.

4.2 P and NP

1. **P** is the class of all yes/no problems which can be solved on a TM in time which is polynomial in n, the size of the input.

   P is closed under polynomial-time reductions:

   (a) A **reduction** R from Problem A to Problem B is an algorithm which takes an input x for A and transforms it into an input $R(x)$ for B, such that the answer to B with input $R(x)$ is the answer to A with input x.

   (b) The algorithm for A is just the algorithm for B *composed* with the reduction R.

   Problem A is *at least as hard as* Problem B if B reduces to it (in polynomial time): If we can solve A, then we can solve B. We write:

$$A \geq_{\mathrm{P}} B \quad \text{or simply} \quad A \geq P.$$

2. **NP** is the class of yes/no problems whiched can be solved in an NTM in (nondeterministic) time polynomial in n, the length of the input.

   Alternatively, it is the class of yes/no problems such that if the answer on input x is yes, then there is a short (polynomial-length in $|x|$) **certificate** that can be checked efficiently (in polynomial time in $|x|$) to prove that the answer is correct.

   - Examples: Compositeness, 3-Sat are in NP.
   - The complement of an NP problem may not be in NP: e.g. if P $\neq$ NP, then Not-satisfiable-3-SAT is not in NP.

3. **NP-complete**: In NP, and all other problems in NP reduce to it.

   That is, A is NP-complete if

   $$A \in \mathrm{NP} \quad \text{and} \quad A \geq_{\mathrm{P}} B \; \forall B \in \mathrm{NP}.$$

   NP-hard: All problems in NP reduce to it, but it is not necessarily in NP.

   - NP-complete problems: circuit Sat, 3-Sat, integer linear programming, independent set, vertex cover, clique.

4. Remember: while we strongly believe so, we don't know whether P $\neq$ NP!

Exercise. *Here is cautionary tale about algorithm runtimes:*

1. *Show that there is a polynomial-time algorithm A_2 for determining whether a Boolean formula has a satisfying truth assignment in which exactly* 2 *of the Boolean variables are true.*
2. *Show that there is a similar polynomial-time algorithm A_k for any $k \geq 0$, which determines whether a formula can be made true by making exactly k of its variables true.*
3. *Then why isn't the following procedure a polynomial-time algorithm for* SAT*? "Given a formula F with n variables, sequentially apply A_0, A_1, ..., A_n to F and accept iff one of the A_i accepts."*

Solution.
We have:

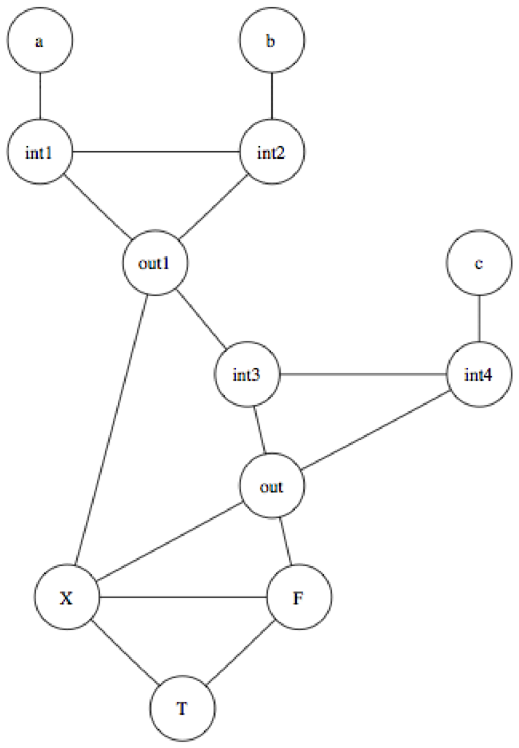
- There are a total of $\binom{n}{2}$ such assignments, so we can try each of them in polynomial time and see whether it is a satisfying assignment.
- Similarly to the previous part, there is a total of $\binom{n}{k}$ such assignments. Since this is polynomial in n, we can again try each assignment and see whether it satiesfies the formula.
- Adding together all the previous, this involves trying $\binom{n}{0} + \ldots + \binom{n}{n} = 2^n$ such assignments, which is no longer doable in polynomial time. In particular, note that while $\binom{n}{k}$ is polynomial in n for constant k, it is exponential in k. Thus, for example, $\binom{n}{n/2} > \frac{2^n}{n}$ is exponential in n.

Exercise. *Here is another NP-complete problem:*

3-Color $= \{G : G$ *is a graph with a valid 3-coloring (no two adjacent vertices have the same color)*$\}$

Here is a reduction from 3-SAT *to* 3-COLOR*:*

1. *Root vertex X. Form a triangle with vertices X, T, F.*
2. *For every variable a in the 3-SAT problem, form a triangle with vertices $X, a, \bar{a}$. Notice that any valid 3-coloring of this graph so far will give a truth assignment to each variable.*
3. *Each clause $a \vee b \vee c$ looks like the figure given.*



Why does this work?

Solution.
Remember that a, b, c are all connected to X so they get some truth assignment.

If a, b are both colored true then out1 will have to be colored true. Likewise if both are colored false then out1 will have to be colored false. The only tricky part is when only one of a, b are colored true, e.g., if a is colored true and b is colored false. Since out1 is connected to X it has to have a truth assignment, so as a result int1, int2 will be colored either false, X (respectively) or X, true (respectively). In the first case out1 will be colored true, in the second case false.

If out1 is colored true, a similar line of reasoning will show that out will in fact be colored true, in which case we say that the clause is satisfied. If out1 is colored false and c is colored true, then out will again be colored true. Finally, there is no valid 3-coloring if out1 and c are both false, but we could have made out1 colored true so we do have a valid 3-coloring in which out gets colored true.

Finally, by what we have shown above, it should be clear that if a, b, c are all false, there is no valid 3-coloring.

Exercise. *Consider the following language:*

$$\text{Subgraph isomorphism} : \{\langle G, H\rangle : G \textit{ contains a subgraph isormorphic to } H\}.$$

(An isomorphism of graphs $G' = (V_1, E_1)$ and $H = (V_2, E_2)$ is a 1-1 function $f : V_1 \to V_2$ such that the map $(u, v) \mapsto (f(u), f(v))$ is a 1-1 function $E_1 \to E_2$.)

Show that Subgraph isomorphism *is NP-complete.*

Solution.
Clearly, in NP. The certificate is just the isomorphism function, which we can easily check in polynomial time.

To show that it is NP-hard, we can reduce from Clique: there is a clique of size $\geq K$ if and only if there is a subgraph isomorphic to C_K, the complete graph on K vertices.

Exercise. *Consider the following problem, which you may assume to be NP-complete:*

3-Exact-Cover : $\{\langle X, \mathcal{C}\rangle : \mathcal{C}$ *a collection of 3-element subsets of* X *containing an exact cover of* $X\}$.

(An exact cover is the same as a partition.)

Show that the analogously defined 4-Exact-Cover *is also NP-complete.*

Solution.
Clearly, 4-Exact-Cover $\in$ NP, since given a subset of $\mathcal{C}$, we can check whether it forms a partition of X.

We show that it is NP-hard by reduction from 3-Exact-Cover: expand X to X' by adding the elements $e_1, \ldots, e_{|X|/3}$. For each 3-element set $S \in \mathcal{C}$, define 4-element sets $S(i) = S \cup \{e_i\}$ for $1 \leq i \leq q$. Solving 4-Exact-Cover for X' with

$$\mathcal{C}' = \{S(i) : S \in \mathcal{C}, 1 \leq i \leq q\}$$

will give the result for 3-Exact-Cover.

4.3 Decidability and Recognizability

A Turing Machine M **decides** a language L if it halts on every input and L is the set of words for which M has output 1. A language is decidable if there is a TM for which this is the case.

A Turing Machine M **recognizes** a language L, where L is the set of words for which M has output 1. A language is recognizable (equivalently, **recursively enumerable**) if there is a TM for which this is the case.

Some useful techniques for determining decidability are

- Diagonlization: for example, we can show that the languages

  $$A_{TM}, A_{WR} = \{\langle M, w\rangle : M \text{ accepts the input } w\},$$

  where M is either a TM or a word-RAM, respectively, are not decidable by constructing a word-RAM/TM which has to differ from every other TM on at least one input.

- Reductions: if $L_1 \leq_m L_2$ and L_1 is undecidable, then so is L_2; contrapositively, if L_2 is decidable, then so is L_1.

- *Rice's Theorem*: let $\mathcal{P}$ be any subset of the class of r.e. languages such that $\mathcal{P}$ and its complement are both nonempty. Then the language $L_{\mathcal{P}} = \{\langle M\rangle : L(M) \in \mathcal{P}\}$ is undecidable.
  - Important: you can only apply Rice's theorem to the language accepted by a TM, not to any properties of how the TM performs its computation.

We also know that $\textsc{Halt}_{TM}$ is undecidable (reduce from A_{TM}, i.e. if M halts and rejects, instead enter an infinite loop, etc.) and that $\overline{A_{TM}}, \overline{A_{WR}}$ are unrecognizable (acceptance is undecidable but recognizable, so it can't be co-recognizable).

Exercise. *Define a* Stay-Still TM (SSTM) *to be like an ordinary TM, but with an additional possibility of staying still in a transition (instead of being required to move left or right in each step).*

Prove that the language $L = \{\langle M\rangle : M \text{ is an SSTM that stays still in at least one step when run on } \varepsilon\}$ *is undecidable.*

Solution.
As usual in situations of that sort, we reduce from the halting problem on the empty string.

Notice that we can't apply Rice's theorem here!

Exercise. *Prove that it is undecidable whether a given Turing machine recognizes a regular language.*

Solution.
Here we *can* apply Rice's theorem. Observe that there exist non-regular languages which can be recognized by a TM, e.g. $\{a^n b^n : n \geq 0\}$. Thus, the subset $\mathcal{P}$ of regular r.e. languages is both non-empty and not the entire set of r.e. languages. Hence, the language in question is undecidable.

Exercise. *Recall that an NFA can have computations that "die off" because no transitions are applicable from the current state and input symbol.*

Define

$$L_1 = \{\langle N, w\rangle : \text{The NFA } N \text{ has a computation on input } w \text{ that "dies off"}\}.$$

Is L_1 decidable? Is L_1 in P?

Solution.
This is decidable: we can simply check all $\leq |Q|^{|w|}$ possible computation paths of N on w.

If we do the checking in a smarter way, we can also see this is in P. This involves simulating the subset construction locally on w - so we don't build the entire DFA from the subset construction, but rather keep the current state from that DFA we're in, and build the next one from that. Each set has size $\leq |Q|$, and we have to do $\leq |Q|^2$ checks to find the next set, so forming the next step can be done in polynomial time. We will have to do that at most $|w|$ times, giving us a $|Q|^2|w|$ algorithm, which is polynomial in $|\langle N, w\rangle|$. Then, we look for a situation in which some of the states in the current set has no transitions applicable under the next input symbol from w, which can be checked efficiently along the way.

Exercise. *Similarly, an NTM can have computations that "die off" because no transitions are applicable from the current state and tape symbol.*

Define

$$L_2 = \{\langle M, w\rangle : \textit{The NTM } M \textit{ has a computation on input } w \textit{ that "dies off"}\}.$$

Is L_2 decidable? Is L_2 in P?

Solution.
This is undecidable. We can reduce from the halting problem on ε. Given a Turing machine M, we can modify it so that when it's about to halt, it instead enters the first of several new states which have the following effect: write a σ, go right, go back left, and then die off upon reading σ. Here σ is any symbol from M's alphabet.

Letting the new machine be M', the reduction is $\langle M\rangle \mapsto \langle M', \varepsilon\rangle$.

As this is an undecidable problem, it cannot be in P.

4.4 Putting it Together

Exercise. *Fill the blank entries of the following table with* `YES`*,* `NO`*, or* `??` *("currently unknown").*

Language:	*regular*	*decidable*	*r.e.*	*P*	*NP*
$\{\langle N\rangle : N$ *is an NFA that accepts abbababa*$\}$	■				
$\{w : w$ *contains abbababa*$\}$					
$\{\langle M\rangle : M$ *is a TM that accepts abbababa*$\}$					
$\{\langle\varphi\rangle : \varphi$ *is a satisfiable boolean formula*$\}$	■				

Solution.

Language:	regular	decidable	r.e.	P	NP
$\{\langle N\rangle : N$ is an NFA that accepts *abbababa*$\}$	■	YES	YES	YES	YES
$\{w : w$ contains *abbababa*$\}$	YES	YES	YES	YES	YES
$\{\langle M\rangle : M$ is a TM that accepts *abbababa*$\}$	NO	NO	YES	NO	NO
$\{\langle\varphi\rangle : \varphi$ is a satisfiable boolean formula$\}$	■	YES	YES	??	YES

Exercise. *Fill the blank entries of the following table with* YES, NO, *or* ?? *("currently unknown"). No explanations needed. In the following table, M always stands for a Turing machine and D always stands for a DFA.*

Language:	*decidable*	*r.e.*	*co-r.e.*	*P*	*NP*
$\{\langle D_1, D_2\rangle : L(D_1) \subseteq L(D_2)\}$					
$\{\langle M_1, M_2\rangle : L(M_1) \subseteq L(M_2)\}$		■	■		
$\{\langle M, D\rangle : L(M) \subseteq L(D)\}$					

Solution.

Language:	decidable	r.e.	co-r.e.	P	NP
$\{\langle D_1, D_2\rangle : L(D_1) \subseteq L(D_2)\}$	YES	YES	YES	YES	YES
$\{\langle M_1, M_2\rangle : L(M_1) \subseteq L(M_2)\}$	NO	■	■	NO	NO
$\{\langle M, D\rangle : L(M) \subseteq L(D)\}$	NO	NO	YES	NO	NO

For the first row, one can decide whether a regular language $L(D_1)$ is inside another regular $L(D_2)$ by efficiently computing the DFA D_3 deciding $L(D_1)\backslash L(D_2)$ and checking whether this is empty. To construct D_3, construct the DFA whose language is $A \times B$, and of the final states (a, b) of this DFA, only set those for which a is a final state of D_1 and b is not a final state of D_2 to be final states. Whether the DFA for this language is empty can be decided in polynomial time.

For the second row, it suffices to show that the language is undecidable. To do this, reduce from the language $\{\langle M\rangle :\ L(M)\ \text{ is empty}\}$, which we know is undecidable by Rice's theorem.

For the third row, the proof of undecidability is the same as above. To show it is not r.e., reduce from $\overline{A_{TM}} = \{(\langle M\rangle, x) : x \notin L(M)\}$: send (M, x) to (M, D) where D is the DFA with language $\{0,1\}^*\backslash\{x\}$. $x \in L(M)$ iff $L(M) \subseteq \{0,1\}^*\backslash\{x\}$.

Exercise. *Suppose that aliens land on Earth, presenting humans with a black box B that solves* HALT$_{TM}$ *in a single step.*

1. *Describe a method that uses the black box to decide the language $L = \{M : L(M) \neq \emptyset\}$. (Hint: Given M, cleverly create an M' that can be fed into the black box B.)*
2. *Prove that every problem in NP can be solved in polynomial time by a Turing machine with access to the black box.*

Solution.
We have:

1. Given M, define a Turing machine M' which ignores the input and runs M on all strings (say, in lexicographic order), looking for an acceptance and halting if so. Here, note that as when we created an enumerator for a recognizable language, we cannot fully run a computation for a string, because M might never halt on the first string tried. Thus, we first try M for one step on the first string, then for two steps on each of the first two strings, and so on.

   Observe that $L(M) \neq \emptyset$ if and only if M' ever halts, on any input. Thus, we have a reduction $L \leq_P \textsc{Halt}_{TM}$, so we can use the black box on M' and decide L.

2. Given a problem $L \in \text{NP}$, we can reduce membership of x in L in polynomial time in $|x|$ to a SAT instance ϕ. Then, we can define a Turing machine M that discards its input and looks for a satisfying assignment of ϕ by trying all possibilities, accepting if any of them works, and entering an infinite loop otherwise. Notice that producing M can be done in polynomial time, as it requires a description of ϕ and some constant amount of instructions in addition to that. Then M halts if and only if $x \in L$, and we can use the black box to test that in a single step.

5 Algorithmic Toolkit

5.1 Greedy

Greedy algorithms involve, broadly speaking, searching the space of solutions by only looking at the local neighborhood and picking the 'best' choice, for some metric of best.

This is a very useful paradigm for producing simple, fast algorithms that are even correct from time to time, even though they feel a lot like heuristics. It is surprising how well greedy algorithms can do sometimes, so the greedy paradigm is a reasonable thing to try when you're faced with a problem for which you don't know what approach might work (e.g., like on a final exam).

Exercise. *Suppose there you are taking a weird final exam in which there are n problems, and the i-th problem is handed to you at time s_i, and you have t_i minutes to work on it. Moreover, you can't work on any two problems simultaneously. If all problems are worth the same and you know you can solve each one of them in the alloted time, how do you efficiently find a set of problems to solve that will maximize your score on the final, given s_i and t_i?*

Solution.
The solution is to sort the problems by the time you'll *finish* them and then choose them greedily, i.e. always pick the first (according to the sorting) problem possible. So define $f_i = s_i + t_i$ to be the final times for each problem.

Observe that any solution (i.e., a choice of problems satisfying the constraints) can be changed to one in which the first problem chosen is the one with smallest f_i. This is true because we can safely exchange the first problem in such a solution with the one with smallest f_i. Then analogous reasoning applies to show that we can also exchange the second problem in an optimal solution with the second problem the greedy approach would have chosen. We keep going this way and see that the number of problems in any solution is at most the number of problems in the greedy solution.

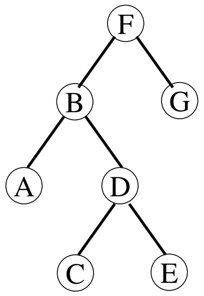
Exercise. *Suppose we are building a binary search tree on a set of data. Normally we would think to make the tree balanced to minimize the search time. But this is not best if we know something about the probability that a node is accessed.*

For example, suppose we are building a tree on characters A,B,C,D,E,F, and G. Our tree must satisfy binary tree restrictions: each node can have at most 2 children, children to the left of a node must be earlier in the alphabet than that node, and children to the right of a node must be later in the alphabet than that node,

Suppose the characters have the following frequencies:

Node	*A*	*B*	*C*	*D*	*E*	*F*	*G*
Prob.	*0.2*	*0.25*	*0.05*	*0.1*	*0.05*	*0.3*	*0.05*

The depth *of the root of the tree is 1; the depth of its children are 2, and so on. The average number of comparisons needed to find an element, which we seek to minimize, is obtained by summing over the product of the depths and probabilities.*



For example, in the tree above, the average number of comparisons needed is

$$0.3 \cdot 1 + 0.25 \cdot 2 + 0.05 \cdot 2 + 0.2 \cdot 3 + 0.1 \cdot 3 + 0.05 \cdot 4 + 0.05 \cdot 4 = 2.2$$

This tree was obtained using a greedy scheme; take the highest probability item for the root, then the highest probability nodes for the children of the root (that give a legal tree!), etc. Is this tree optimal?

Solution.
No. For example, consider the balanced binary search tree on these seven letters. Then, the average number of comparisons needed is

$$0.1 \cdot 1 + 0.25 \cdot 2 + 0.3 \cdot 2 + 0.2 \cdot 3 + 0.05 \cdot 3 + 0.05 \cdot 3 + 0.05 \cdot 3 = 1.95,$$

which is better than the tree given by the greedy algorithm.

5.2 Divide and Conquer

The divide and conquer paradigm is another natural approach to algorithmic problems, involving three steps:

1. Divide the problem into smaller instances of the same problem;
2. Conquer the small problems by solving recursively (or when the problems are small enough, solving using some other method);
3. Combine the solutions to obtain a solution to the original problem.

Examples we've seen in class include binary search, matrix multiplication, and fast integer multiplication. Because of the recursion plus some additional work to combine, the runtime analysis of divide and conquer algorithms is often easy if we use the Master theorem.

Exercise. *You're given an array A of n distinct numbers that is sorted up to a cyclic shift, i.e. $A = (a_1, a_2, \dots, a_n)$ where there exists a k such that $a_k < a_{k+1} < \dots < a_n < a_1 < \dots < a_{k-1}$, but you don't know what k is. Give an efficient algorithm to find the largest number in this array.*

Solution.
Initialize $i = 1$, and double it until $A[i] < A[1]$. Then you know that a_n must be somewhere between $A[i/2]$ and $A[i]$. Then use binary search between $A[i/2]$ and $A[i]$: if $A[j] < A[1]$, you know you're to the right of a_n, and otherwise you're to the left. This algorithm will take $O(\log n)$ time.

5.3 Dynamic Programming

The important steps of dynamic programming are:

1. Define your subproblem.
2. Define your recurrence relation.
3. Define your base cases, and how you will fill up the subproblem matrix.
4. Analyze the asymptotic runtime and memory usage.

Exercise. *Given a matrix consisting of 0's and 1's, find the maximum size square sub-matrix consisting of only 1's.*

Solution.
Let the binary matrix be M with dimensions $m \times n$. We can then construct a second matrix S where $S[i, j]$ corresponds to the size of the square submatrix containing $M[i, j]$ in the right-most and bottom-most corner. Note that if $M[i, j] = 1$, then $S[i, j] = 0$.

We can then copy the first row and column from M to S because at most it will be a square of size

1×1. We can then fill S according to

$$S[i,j] = \begin{cases} \min\left(S[i,j-1], S[i-1,j], S[i-1,j-1]\right) + 1 & : M[i,j] = 1 \\ 0 & : M[i,j] = 0 \end{cases}$$

This will take $O(mn)$ time and space.

Exercise. *There are n rooms on the hallway of an unnamed Harvard house. A thief wants to steal the maximum value from these rooms, but he/she cannot steal from two adjacent rooms because the owner of a robbed room will warn the neighbors on the left and right. What is the largest possible value of stolen goods?*

Solution.
Let the value of room i be v_i. We can define $f(i)$ to be the cumulative maximal value of stolen goods from rooms $1, \ldots i$. Our recurrence then becomes (with base case $f(0) = v_0, f(1) = \max(v_0, v_1)$)

$$f(i) = \max[v_i + f(i-2), f(i-1)]$$

This works because the first number gives the maximum value of stolen goods if the ith room is robbed, and the second number gives the maximum value of stolen goods if the ith room is not robbed.

This solution would take $O(n)$ because we are building only a 1-d array.

Exercise. *Based on your advanced knowledge of computer science theory, you open a high-tech consulting firm. Based on the work available, you can spend each month in Boston or San Francisco; to manage your leases and other expenses, you work on a month to month basis. You have enough work lined up to choose your schedule in advance and must choose a location for each month.*

Suppose the months are numbered from 1 to n. You know that in month i you can earn B_i in Boston and S_i in San Francisco. However, each time you switch cities, you incur a fixed cost M of moving from one side of the country to the other. Given the values of the B_i's and S_i's, you want to maximize earnings minus costs.

1. *Show that the greedy algorithm of choosing the city where you earn the most each month is not optimal by giving an example.*
2. *Give a dynamic programming algorithm that determines the city for each month.*

Solution.
We have:

1. Suppose $n = 2$, and let $B_1 = S_2 = M, B_2 = S_1 = M/2$. Then the greedy algorithm would have you in Boston on month 1 and in San Francisco in month 2, for a net profit of $M/2$.

Meanwhile, just staying in either city for the entire time would allow you to earn $3M/2$.

2. We will fill two n-entry arrays, B and S. In particular, $B[i]$ will be the maximum possible net earnings in months 1 to i, if i is spent in Boston; $S[i]$ is defined analogously for San Francisco. Then, we have that $B[0] = S[0] = M$, and that

$$B[i] = B_i + \max(B[i-1], S[i-1] - M)$$
$$S[i] = S_i + \max(S[i-1], B[i-1] - M).$$

We can clearly compute this in both linear time and space.

5.4 Disjoint-set Data Structure.

The disjoint-set data structure enables us to efficiently perform operations such as placing elements into sets, querying whether two elements are in the same set, and merging two sets together. To make it work, we must implement the following operations:

(i) MAKESET(x) — create a new set containing the single element x.

(ii) UNION(x, y) — replace sets containing x and y by their union.

(iii) FIND(x) — return the name of the set containing x.

We add for convenience the function LINK(x, y) where x, y are roots: LINK changes the parent pointer of one of the roots to be the other root. In particular, UNION(x, y) = LINK(FIND(x), FIND(y)), so the main problem is to make the FIND operations efficient.

Exercise. *What are the two main methods of optimization for disjoint-set data structures? Show an example of each.*

Solution.
The two main methods are:

1. **Union by rank.** When performing a UNION operation, we prefer to merge the shallower tree into the deeper tree.
2. **Path compression.** After performing a FIND operation, we can simply attach all the nodes touched directly onto the root of the tree.

5.5 Linear Programming

1. *Simplex Algorithm*: The geometric interpretation is that the set of constraints is represented as a polytope and the algorithm starts from a vertex then repeatedly looks for a vertex that is adjacent and has better objective value (its a hill-climbing algorithm). Variations of the simplex algorithm are *not* polynomial, but perform well in practice.

2. Standard form required by the simplex algorithm: *minimization, nonnegative variables and equality constraints.*

Exercise. *Suppose that we have a general linear program with n variables and m constraints and suppose we convert it into standard form. Give an upper bound on the number of variables and constraints in the resulting linear program.*

Solution.
Converting from a maximization to minimization does not add any variables or constraints. When ensuring that all constraints are equality constraints, one needs to introduce at most m new slack variables (one for each inequality constraint). The slack variables are added in a way that they are nonnegative. To ensure that all n variables are nonnegative, one substitutes x for $x^+ - x^-$, where $x^+, x^- \geq 0$ for each appearance of unrestricted variable x and multiply by -1 each non-positive one. So at most n variables are added. Thus the resulting LP has at most $2n + m$ variables and m constraints.

6 Algorithmic Problems

6.1 Sorting

Sorting is an extremely basic task that arises in a variety of contexts, which warrants the existence of many different algorithms for sorting that make various assumptions about the data and have various nice properties. The serious sorting algorithms we've seen so far include bubble sort, merge sort, counting sort, bucket sort, radix sort, and van Emde Boas trees. The non-serious one is Stoogesort.

Exercise. *Given an array of n integers, show an $O(n^2 \log n)$ and an $O(n^2)$ algorithm to check whether there is a triple of distinct numbers in the array that sum to zero.*

Solution.
To obtain an $O(n^2 \log n)$ algorithm, create the array of all distinct pairs of integers (x_i, x_j) with $j > i$, and then sort this array based on the value $x_i + x_j$. Now, we can iterate through the original array, and if we see a number a, look for $-a$ in the array of sums of pairs via binary search. Making sure we look at triples of distinct numbers can be easily done, e.g. by remembering the pair each sum came from. These steps run in time $O(n^2 + n^2 \log n^2 + n \log n^2) = O(n^2 \log n)$ time.

To obtain a better runtime of $O(n^2)$, first, sort the original array, and call the resulting values $x_1, \ldots, x_n$. Then, for each i, we want to test whether there exist $x_i + x_j + x_k = 0$ with i, j, k distinct. To to this, start by letting $j = 1$ and $k = n$. Then at each point, if $x_i + x_j + x_k < 0$, increment j; alternatively, if $x_i + x_j + x_k > 0$, decrement k (as we move j, k, skip i). If $j = k$, then move on to the next value of i. Clearly, this runs in time $O(n^2)$, since it requires time $O(n)$ for each i. To show that this works, suppose that on the correct i, without loss of generality j reaches the right value before k; then, because the $x's$ are sorted, we will have $x_i + x_j + x_k > 0$ until k also reaches the right value.

Exercise. *Suppose we adopt an algorithm similar to Stoogesort, with the following change: instead of sorting the first two-thirds of the list recursively, use merge sort. What is the running time?*

Solution.
Because mergesort requires time $O(n \log n)$, the new runtime recursion is

$$T(n) = T(2n/3) + cn \log n$$

for some constant c.

Now, note that $n \log n = O(n^{1+\epsilon})$ for all $\epsilon > 0$. Consequently, by the master theorem, because $1 < (3/2)^{1+\epsilon}$, $T(n) = O(n^{1+\epsilon})$ for all ϵ. To show that in fact $T(n) = \Theta(n \log n)$ (and not, say $\Theta(n \log^2 n)$), note that

$$T(n) = c\left(n \log n + \frac{2n}{3} \log \frac{2n}{3} + \dots\right) \leq c\left(n + \frac{2n}{3} + \dots\right) \log n = 3cn \log n.$$

Exercise. *Show that there is no comparison sort whose running time is linear in at least half of the* $n!$ *inputs of length* n *(that is to say, there is no comparison-based sorting algorithm and polynomial* $p(n)$ *such that for all* n*, the algorithm takes time at most* $p(n)$ *for at least half of the* $n!$ *possible inputs).*

What about $1/n$ *of the inputs? Or* $1/2^n$*?*

Solution.
Suppose that we have some such sorting algorithm with at most $p(n)$ comparisons on half of the possible inputs. Then, on these inputs, there are at most $2^{p(n)}$ sets of possible results which it can obtain on its comparisons. Consequently, because there are $\frac{n!}{2}$ possible inputs, we must have

$$2^{p(n)} \geq \frac{n!}{2} \Rightarrow p(n) \geq \log_2(n!/2) = \log_2(n!) - 1 = \Theta(n \log n),$$

so $p(n)$ cannot be linear.

Similarly, for $1/n$ of the inputs, we must have

$$2^{p(n)} \geq \frac{n!}{n} = \log_2((n-1)!) = \Theta(n \log n),$$

so $p(n)$ cannot be linear.

Finally, for $1/2^n$ of the inputs, we have

$$\begin{aligned}
2^{p(n)} &\geq \frac{n!}{2^n} \\
&= \log_2(n!/2^n) \\
&= \log_2(n/2) + \log_2((n-1)/2) + \ldots + \log_2(1/2) \\
&> \log_2(\lfloor n/2 \rfloor) + \log_2(\lfloor n/2 \rfloor - 1) + \ldots \\
&= \Theta\left(\frac{n}{2}\log\frac{n}{2}\right),
\end{aligned}$$

which makes it again impossible for $p(n)$ to be linear.

6.2 Graph Traversal

Graph traversals are to graphs as sorting algorithms are to arrays - they give us an efficient way to put some notion of order on a graph that makes reasoning about it (and often, making efficient algorithms about it) much easier.

Perhaps the best example of that analogy is the topological sort of a directed acyclic graph, which orders the vertices by an ancestor-descendant relation. This simplifies the execution of many algorithms on such graphs. It is an example application of *depth-first search* (DFS), which we've also seen is useful for detecting cycles in time $O(|V| + |E|)$ and for connected components.

The other graph traversal paradigm we've seen is *breadth-first search* (DFS), which is useful for calculating shortest paths.

Remember: the basic distinction between the two graph traversal paradigms is that DFS keeps a **stack** of the vertices (you can think of it as starting with a single vertex, adding stuff to the right, and removing stuff from the right as well), while BFS keeps a **queue** (which starts with a single vertex, adds stuff to the right, but removes stuff from the *left*) or a **heap** (which inserts vertices with a priority, and then removes the lowest priority vertex each time).

Exercise. *Given a connected graph G, determine in $O(|V| + |E|)$ time if it's bipartite (and if it is, find a partition that establishes that).*

Solution.

Lemma 1.
G is bipartite if and only if it is 2-colorable.

Proof. If G is bipartite, then we can color the vertices on each side in a distinct color. Meanwhile, if it is 2-colorable, then the vertices of each color can form a side of the partition. □

Given the lemma above, it suffices to check if G is 2-colorable. Consequently, run a DFS on G, coloring each node's unvisited children the opposite color of that node. Finally, for each edge, check that the two vertices on both sides are different colors; G is 2-colorable if and only if this is true for each edge. If there is a 2-coloring, then every vertex's children must be the opposite color from that vertex, so this procedure must find it.

The above runs in time $O(|V| + |E|)$, as desired.

Another solution would be to use BFS, and label all the vertices that are an odd distance from the source as red and all the vertices that are an even distance from the source blue. If G is bipartite, the two parts should be the blue and the red vertices - so to finish the algorithm, check if there are any edges running between two blue vertices or two red vertices. This can also be done in $O(|V| + |E|)$ time.

Exercise. *Running DFS on a graph will separate it into a set of trees – one for each time we restart the DFS algorithm at a new vertex. Explain how a vertex u of a directed graph G can end up in a depth-first tree containing only u, even though u has both incoming and outgoing edges in G.*

Solution.
Consider the graph

$$v_1 \to v_2 \to v_3.$$

Then, if we start our DFS at v_3, then restart at v_2, then restart at v_1, we will be left with three separate trees of a single vertex each.

6.3 Minimum Spanning Trees

A **tree** is an undirected graph $T = (V, E)$ satisfying all of the following conditions:

1. T is connected,
2. T is acyclic,
3. $|E| = |V| - 1$.

However, any two conditions imply the third.

A **spanning tree** of an undirected graph $G = (V, E)$ is a subgraph which is a tree and which connects all the vertices. (If G is not connected, G has no spanning trees.)

A **minimum spanning tree** is a spanning tree whose total sum of edge costs is minimum.

Exercise. *What are two efficient algorithms for finding the minimum spanning tree, and when would you use either?*

Solution.
The two algorithms are as follows:

1. **Prim's algorithm.** We choose a single vertex and then grow the tree one edge at a time by selecting the minimum weight edge that connects to vertices not yet in the tree. Thus we are always adding the "closest" vertex to the tree. The implementation of this is almost identical to that of Dijkstra's algorithm. Depending on the minimum edge weight data structure, this can take asymptotically anywhere from $O(|V|^2)$ using an adjacency matrix or $O(|E| + |V| \log |V|)$ using an Fibonacci heap.

2. **Kruskal's algorithm.** Sort the edges. Repeatedly add the lightest edge that doesn't create a cycle, until a spanning tree is found.

   Notice that in Prim's we explicitly set S based on X, whereas here we implicitly "choose" the cut S after we find the lightest edge that doesn't create a cycle. In other words, we don't actually find a cut corresponding to the edge e to be added, but we know that one must exist — if not, adding e would create a cycle. Kruskal's algorithm runs in $O(E \log V)$ time.

If you have a dense graph with more edges, Prim's might be advantageous to use, but Kruskal's simper data structures and sufficiently fast runtime makes it a good choice for typical (sparse) graphs.

6.4 Shortest Paths

The shortest paths problem and various variants are very natural questions to ask about graphs. We've seen several shortest paths algorithms in class:

1. *Dijkstra's algorithm*, which finds all single-source shortest paths in a directed graph with non-negative edge lengths, is an extension of the idea of a breadth-first search, where we are more careful about the way time flows. Whereas in the unweighted case all edges can be thought of taking unit time to travel, in the weighted case, this is not true any more. This necessitates that we keep a *priority queue* instead of just a queue.

2. A single-source shortest paths algorithm (*Bellman-Ford*) that we saw for general (possibly negative) lengths consists of running a very resonable local update procedure enough times that it propagates globally and gives us the right answer. Recall that this is also useful for detecting negative cycles!

3. A somewhat surprising application of dynamic programming (*Floyd-Warshall*) that finds the shortest paths between all pairs of vertices in a graph with non-negative weights by using subproblems $D_k[i, j]$ = shortest path between i and j using intermediate nodes among $1, 2, \ldots, k$.

Exercise. *Show how to modify the Bellman-Ford algorithm so that instead of $|V| - 1$ iterations, it performs $\leq m + 1$ iterations, where m is the maximum over all $v \in V$ of the minimum number of edges in a shortest (in the weighted sense) path from the source s to v.*

Solution.
During each new iteration, also keep track of whether any of the `dist` values changed. If all of them

ever remain the same, stop. From our discussion in class, it follows that each path needs at most m updates, and after that it will stop updating. Thus, we will detect that in $m+1$ iterations.

Exercise. *Suppose we're given a weighted directed graph $G=(V,E)$ in which edges that leave the source s may have negative weights, but all other edges have non-negative weights, and there are no negative weight cycles. Prove that Dijkstra's algorithm still correctly finds the shortest paths from s in this graph.*

Solution.
Let's compare the execution of Dijkstra's algorithm on G to the execution on a graph G' where we've added the same amount m to the weights of all edges coming out of the source so as to make them non-negative.

On the one hand, since all edge weights in G' are non-negative, we know that Dijkstra's algorithm will find the correct shortest-path distances in G'.

On the other hand, observe that the flow of Dijkstra's algorithm is controlled by two things: the comparisons $\texttt{dist}[w] > \texttt{dist}[v] + \texttt{length}[v,w]$, and the element with lowest priority in the heap.

Observe that at any step in Dijkstra's algorithm, in the comparison $\texttt{dist}[w] > \texttt{dist}[v]+\texttt{length}[v,w]$ we're actually comparing the lengths of two paths from s w. Whenever $w \neq s$, both of these paths contain exactly one edge coming out of s. So the truth value of the comparison will be the same in G and G' because the difference is m on both sides of the comparison. Moreover, once we pop s from the queue, the queue for G' will be a copy of the queue for G, but with all priorities shifted up by m. This preserves the minimal element.

It follows that Dijkstra's algorithm is the same on G and G', up to the priorities of all vertices $v \neq s$ shifted by m. Thus, Dijkstra's algorithm on G will find the shortest paths in G'. But the shortest paths in G' to all $v \neq s$ are the same as those in G by an argument similar to the above: if we have a path $p : s \to v$ which is shortest in G', this means that $l'(p) \leq l'(q)$ for any $q : s \to v$, where l' is distance calculated in G', and l is distance calculated in G. But then, $l(p) = l'(p) - m \leq l'(q) - m = l(q)$ for all such q in G, hence p is a shortest path in G.

Finally, it remains to show that we find the right distance to s itself in G - which is clear because we assumed G has no negative cycles, so the true distance is 0, and this is what Dijkstra's algorithm gives.

6.5 Network Flows

1. **Maximum flow** is equal to **minimum cut**.
2. *Ford-Fulkerson algorithm*: to find the maximum flow of a graph with integer capacities, repeatedly use DFS to find an **augmenting path** from start to end node in the residual network (note that you can go backwards across edges with negative residual capacity), and then add that path to the flows we have found. Stop when DFS is no longer able to find such a path.

(a) **Residual flow**: To form the residual network, we consider all edges $e = (u, v)$ in the graph as well as the reverse edges $\bar{e} = (v, u)$.

i. Any edge that is not part of the original graph is given capacity 0.

ii. If $f(e)$ denotes the flow going across e, we set $f(\bar{e}) = -f(e)$.

iii. The **residual capacity** of an edge is $c(e) - f(e)$.

The runtime is $O((m + n)(\text{max flow}))$.

3. Matchings: The maximum size matching in a bipartite graph can be solved by taking the integer maximum flow from one side to the other.

Exercise. *Give a counterexample to the following statement: if all edges have different capacities, then the network has a unique minimum cut.*

Solution.
Consider the graph

$$\begin{tikzcd}
s \arrow[r, "2"] \arrow[dr, "1"'] & v_1 \arrow[r, "5"] & v_3 \arrow[r, "3"] & t \\
 & v_2. \arrow[ur, "4"'] & &
\end{tikzcd}$$

Then, cuts $\{s, v_1, v_2, v_3\}, \{t\}$ and $\{s\}, \{v_1, v_2, v_3, t\}$ are both minimum cuts.

6.6 2-player Games

Exercise. *Consider the following zero-sum game, where the entries denote the payoffs of the row player:*

$$\begin{pmatrix} 2 & -4 & 3 \\ 1 & 3 & -3 \end{pmatrix}$$

Write the column player's maximization problem as an LP. Then write the dual LP, which is the row player's minimization problem.

Solution.
The column player solves

variables	z	x_1	x_2	x_3		
constraints	-1	2	-4	3	$\leq$	0
	-1	1	3	-3	$\leq$	0
		1	1	1	$=$	1
	unr	$\geq$	$\geq$	$\geq$		
objective	1					min

The dual problem, solved by the row player, is

variables	y_1	y_2	$\tilde{z}$		
constraints	1	1		$=$	1
	-2	-1	1	$\leq$	0
	4	-3	1	$\leq$	0
	-3	3	1	$\leq$	0
	$\geq$	$\geq$	unr		
objective			1		max

7 Probabilistic and Approximation Algorithms

7.1 Hashing

A hash function is a mapping $h : \{0, \ldots, n-1\} \to \{0, \ldots, m-1\}$. In most applications, $m < n$.

Hashing-based data structures (e.g. **hash tables, bloom filters**) are useful since they ideally allow for constant time operations (lookup, adding, and deletion). However, the major problem preventing this is collisions (which occur more, for example, if m is too small or you use a poorly chosen hash function).

Exercise. *When n balls are thrown randomly into n bins, the expected fraction of empty bins is about $1/e$. When $2n$ balls are thrown randomly into n bins, the expected fraction of empty bins is about:*

- $1/e$
- $1/e^2$
- $1/\sqrt{e}$
- $2/e$
- $1/(2e)$

Solution.
Because the expected fraction using n balls is about $\frac{1}{e}$, by symmetry and linearity of expectation the probability that any particular bin is empty must be $\frac{1}{e}$. However, we also can compute the probability that it is empty as $\left(\frac{n-1}{n}\right)^n$.

Using this, for $2n$ balls the probability that any given bin is empty is

$$\left(\frac{n-1}{n}\right)^{2n} = \left(\left(\frac{n-1}{n}\right)^n\right)^2 \approx \frac{1}{e^2}.$$

Exercise. *I create a Bloom filter with 1000 bits, using 5 hash functions for each item stored in the Bloom filter. After inserting a bunch of items, I find that 600 bits in the filter are set to 1. Write an expression for the number of items you think were inserted to the filter, explaining how you derived your expression.*

Solution.
Suppose that I have inserted m items. Then, this results in $5m$ hash values. Consequently, assuming perfect hashing, the probability that any given bit of the filter is 0 is $\left(\frac{999}{1000}\right)^{5m}$. Consequently, we want

$$1000 - 600 = 1000 \cdot \left(\frac{999}{1000}\right)^{5m} \approx \frac{1000}{e^{m/200}} \Rightarrow m \approx 200 \ln \frac{5}{2}.$$

7.2 Random Walks

A random walk is an iterative process on a set of vertices V. In each step, you move from the current vertex v_0 to each $v \in V$ with some probability. The simplest version of a random walk is a one-dimensional random walk in which the vertices are the integers, you start at 0, and at each step you either move up one (with probability 1/2) or down one (with probability 1/2).

2-SAT: In lecture, we gave the following randomized algorithm for solving 2-SAT. Start with some truth assignment, say by setting all the variables to false. Find some clause that is not yet satisfied. Randomly choose one of the variables in that clause, say by flipping a coin, and change its value. Continue this process, until either all clauses are satisfied or you get tired of flipping coins.

We used a random walk with a completely reflecting boundary at 0 to model our randomized solution to 2-SAT. Fix some solution S and keep track of the number of variables k consistent with the solution S. In each step, we either increase or decrease k by one. Using this model, we showed that the expected running time of our algorithm is $O(n^2)$.

7.3 Approximation Algorithms

While we don't know how to solve NP-hard problems exactly, we can often get an answer that is reasonably "close" to the optimal.

Some examples include:

1. Vertex cover: Want to find minimal subset $S \subseteq V$ such that every $e \in E$ has at least one endpoint in S.

   To do this, repeatedly choose an edge, throw both endpoints in the cover, delete endpoints and all adjacent edges from graph, continue. This gives a 2-approximation.

2. Max cut:

(a) Randomized algorithm: Assign each vertex to a random set. This gives a 2-approximation in expectation.

(b) Deterministic algorithm: Start with some fixed assignment. As long as it is possible to improve the cut by moving some vertex to a different set, do so. This gives a 2-approximation.

3. MAX-SAT: Asks for the maximum number of clauses which can be satisfied by any assignment. Setting every variable randomly satisfies on expectation $\frac{2^k-1}{2^k}$ for a k-SAT problem where no clause may contain the same variable twice. (This can also be done with a deterministic polynomial-time algorithm.)

Exercise. *Suppose we are given a set of cities represented by points in the plane, $P = \{p_1, \dots, p_n\}$. We will choose k of these cities in which to build a hospital. We want to minimize the maximum distance that you have to drive to get to a hospital from any of the cities p_i. That is, for any subset $S \subset P$, we define the cost of S to be*

$$\max_{1 \le i \le n} \operatorname{dist}(p_i, S) \quad \textit{where} \quad \operatorname{dist}(p_i, S) = \min_{s \in S} \operatorname{dist}(p_i, s).$$

This problem is known to be NP-hard, but we will find a 2-approximation. The basic idea: Start with one hospital in some arbitrary location. Calculate distances from all other cities — find the "bottleneck city," and add a hospital there. Now update distances and repeat.

Come up with a precise description of this algorithm, and prove that it runs in time $O(nk)$.

Solution.
For each city p_i we let d_i denote its distance from the set of hospitals so far. Suppose we have just added the j-th hospital s_j. To decide where to add the next hospital, we look at $O(n)$ cities, updating the distances d_i by taking $d_i = \min(d_i, \operatorname{dist}(p_i, s_j))$. The bottleneck city is the one with maximal d_i after this update, so we set s_{j+1} to be this city. We continue until k hospitals have been added, and we do a last update of the distances to find the final cost of our choice of S.

Exercise. *Prove that this gives a 2-approximation.*

Solution.
Consider an optimal choice of $S^* = \{s_1, \dots, s_k\}$, and let D_j denote the disc with center s_j and radius r^*, where r^* denotes the cost of S^*. Now consider the S that we obtained from our approximation algorithm, which has cost r. If every D_j contains some point of S then we are done, because then every p_i will be within distance $2r^*$ of S. Suppose not: By the pigeonhole principle, two points $s, t \in S$ must be contained in a single disc $D = D_j$. But by the way we constructed S, $\operatorname{dist}(s, t) \ge r$. We also have $\operatorname{dist}(s, t) \le 2r^*$, so again $r \le 2r^*$ which concludes the proof.

Exercise. *Suppose there exists a poly-time algorithm for finding a clique in a graph whose size is at least half the size of the maximal clique. Show that for any constant $k < 1$, there would then exist a poly-time algorithm for finding a clique of size at least k times the size of the maximal clique.*

(In fact, approximating max clique within any constant factor is NP-hard, so it is unlikely that a poly-time constant-factor approximation exists.)

Solution.
We demonstrate an approximation algorithm that is good within a factor of $\frac{1}{\sqrt{2}}$. The approach extends immediately to factors of $\frac{1}{2^{1/m}}$.

The idea is to "square" the size of the maximal clique while polynomially increasing the size of the graph. "Un-squaring" a half-sized clique in this larger graph will yield, in polynomial time, a clique in the original graph that is at least $\frac{1}{\sqrt{2}}$ times as big as the original maximal clique.

Suppose we have a graph G with v nodes and e edges. We transform G into a new graph $G^{(1)}$ by replacing every node with a copy of G. $G^{(1)}$ therefore has v "super-nodes," which are connected to each other by "super-edges." That is, if there is a "super-edge" between super-nodes U and V, then every node in U is adjacent to every node in V. Note that the size of $G^{(1)}$ is polynomial in the size of G.

Suppose the maximal clique in G is size M. Then by construction, the maximal clique in $G^{(1)}$ is at least size M^2: every super-node has a maximally connected subgraph of size M, and there are M super-nodes in $G^{(1)}$ that are maximally connected (via super-edges) in exactly the same way. In time polynomial in the size of $G^{(1)}$ (hence polynomial in the size of G), we can find a clique K' in $G^{(1)}$ of size at least $\frac{M^2}{2}$.

Let x be the largest number of nodes belonging to K' that also belong to a single super-node. Let y be the number of super-nodes that have some intersection with K'. Then $xy \geq \frac{M^2}{2}$, which implies $x \geq \frac{M}{\sqrt{2}}$ or $y \geq \frac{M}{\sqrt{2}}$. Clearly, if $x \geq \frac{M}{\sqrt{2}}$, we have found a clique of size x in G. The key is that if $y \geq \frac{M}{\sqrt{2}}$, we have also found a clique of size y in G: if super-node U and super-node V both have non-empty intersection with K', then U and V must be connected by a super-edge, which implies that u and v in G are connected by an edge. In other words, a clique of "super-nodes" corresponds exactly to a clique of nodes. We have found a clique of size at least $\frac{M}{\sqrt{2}}$, as desired.

8 PCP Theorem, Hardness of Approximation

Recall the complexity class PCP:

- A language L is in the complexity class $\mathrm{PCP}_{c,s}(r(n), q(n))$ if there is a polynomial time randomized verifier V such that for any $x \in \{0,1\}^*$, if we let $n = |x|$, then
  - On input $\langle x, \pi \rangle$, V reads x, tosses $r(n)$ (fair) coins, reads $q(n)$ bits of π, and decides whether to accept or reject.
  - If $x \in L$, then there is $\pi \in \{0,1\}^{poly(n)}$ such that $P[V(\pi, x) = 1] \geq c$ (Here c is called the completeness).
  - If $x \notin L$, then for all $\pi \in \{0,1\}^{poly(n)}$, $P[V(\pi, x) = 1] \leq s$. (Here s is called the soundness.)

- The PCP theorem states that there is a universal constant $q > 0$ such that $NP = PCP_{1,1/2}(O(\log n), q)$.
- We often write PCP(q, r) to denote PCP$_{1,1/2}(q, r)$. In this handout the alphabet is always $\Sigma = \{0, 1\}$.

8.1 Exercises

Exercise. *Show that PCP$(\log n, O(1)) \subseteq$ NP.*

Solution.
Suppose $L \in$ PCP$(\log n, O(1))$, and let V be a verifier for L. Now create a verifier V' that reads π, simulates V on all $2^{\log n} = n$ possibilities for the $\log n$ random coin flips, and accepts if and only if V accepts on all of the simulations. It is clear that V runs in polynomial time and also that V is a valid verifier for L.

Exercise. *Show the following:*

- *PCP$(poly(n), 0) = co - RP$.*
- *PCP$(0, 0) = P$.*
- *PCP$(\log n, O(1)) =$ PCP$(\log n, poly(n))$.*

Solution.
For the first one, take a language L in PCP$(poly(n), 0)$ and consider a verifier V for L. On receiving an input x, note that L flips $poly(n)$ coins and makes a decision to accept or reject (without looking at the proof π). If $x \in L$, we are guaranteed that V will accept with probability 1, and if $x \notin L$, then V will accept with probability at most 1/2. Now consider a verifier V' (for the language $\bar{L}$) that accepts iff V rejects. In other words, if $x \in \bar{L}$, then V' accepts x with probability at least 1/2, and if $x \notin \bar{L}$, then V' accepts x with probability 0. This shows that $\bar{L} \in RP$. Hence $L \in co - RP$. Showing the other direction $co - RP \subseteq PCP(poly(n), 0)$ is nearly identical.

For the second one, note that any verifier for a language L in $PCP(0, 0)$ must be deterministic (since it does not flip any coins), so it is just a polynomial time algorithm that decides L.

For the third one, note that we have the inclusions: $NP \subseteq PCP(\log n, O(1)) \subseteq PCP(\log n, poly(n)) \subseteq NP$. The first inclusion follows from the PCP theorem. The second inclusion follows since any verifier that reads at most $O(1)$ bits by definition reads at most $poly(n)$ bits. The third inclusion follows in nearly the same way as the first exercise above: you can go through the proof of that exercise again and check that it still holds if the verifier is allowed to read $poly(n)$ instead of $O(1)$ bits.

Exercise (Challenge)**.** *You may assume the following fact: Given a constant b, there is a constant k such that for all $n \in \mathbb{N}$, there is a graph H that can be constructed in polynomial time (that is, time $poly(n)$), such that*

1. *H has n^b vertices and the maximum degree is constant (in n).*
2. *For any subset S of the vertices of H, of size at most $n^b/2$, we have*

$$P[\textit{All vertices of a } k\log n\textit{-length random walk on } H \textit{ lie in } S] < 1/n.$$

By random walk, we mean a walk on the vertices of H that at each vertex, picks one of its neighbors uniformly at random and travels to that vertex. Such a graph H is called an expander graph; you do **not** *need to know about expander graphs in this course!*

Using this fact, prove that

$$PCP_{1,1/2}(O(\log n), O(1)) \subseteq PCP_{1,1/n}(O(\log n), O(\log n)).$$

Solution.
Consider a language L in $PCP_{1,1/2}(\log n, 1)$. Let V be a verifier for L: suppose it queries $b \log n$ random bits and queries q bits of the proof π, for some constants b, q. Now, we construct a verifier V' as follows: it constructs the graph H from the fact above (this can be done in time polynomial in n, since b is a constant). We associate each vertex of H with one of the $2^{b\log n} = n^b$ possible results from the random coin flips that B makes.

The verifier V' then constructs a random walk of length $k \log n$ on H. The random walk can be accomplished by flipping $O(\log n)$ coins (in particular, at each vertex of degree d (which is a constant in n), it is not hard to see that flipping $O(\log d) = O(1)$ coins allows one to choose the next vertex uniformly at random; doing this $k \log n$ times leads to flipping $O(\log n)$ coins overall).

At each of the $k \log n + 1$ vertices of H that V' crosses in its random walk, it simulates the verifier V for the outcome of the random coin flips corresponding to that vertex. Then V' accepts iff V accepts on all $k \log n + 1$ simulations.

It is clear that V' runs in polynomial time, flips $O(\log n)$ coins, and queries $O(\log n)$ bits of the proof π (each simulation of V queries $O(1)$ bits, and V' makes $O(\log n)$ such simulations).

Now, if $x \in L$ and π is a proof that makes V accept x with probability 1, then V', given proof π, also accepts x with probability 1.

If $x \notin L$, and if π is any proof, then note that V will accept (given π) on at most $n^b/2$ outcomes of its random coin flips. If S is the corresponding set of vertices of H, then $|S| > n^b/2$, and V' accepts x iff the random walk lies entirely in S. By the fact above, the probability of this happening is at most $1/n$, as desired.