Contents

1 **Notes about the Final** **2**

2 **Mathematics** **2**
- 2.1 Big-O and the Master Theorem . . . 2
- 2.2 Countability . . . 3
- 2.3 Probability . . . 3

3 **Models of Computation** **5**
- 3.1 DFAs & NFAs . . . 5
- 3.2 Turing Machines . . . 6
- 3.3 Word-RAMs . . . 7

4 **Complexity** **7**
- 4.1 Regular Languages . . . 7
- 4.2 P and NP . . . 8
- 4.3 Decidability and Recognizability . . . 10
- 4.4 Putting it Together . . . 12

5 **Algorithmic Toolkit** **14**
- 5.1 Greedy . . . 14
- 5.2 Divide and Conquer . . . 15
- 5.3 Dynamic Programming . . . 16
- 5.4 Disjoint-set Data Structure. . . . 17
- 5.5 Linear Programming . . . 18

6 **Algorithmic Problems** **18**
- 6.1 Sorting . . . 18
- 6.2 Graph Traversal . . . 19
- 6.3 Minimum Spanning Trees . . . 20
- 6.4 Shortest Paths . . . 21
- 6.5 Network Flows . . . 22
- 6.6 2-player Games . . . 23

7 **Probabilistic and Approximation Algorithms** **23**
- 7.1 Hashing . . . 23
- 7.2 Random Walks . . . 24
- 7.3 Approximation Algorithms . . . 24

1 Notes about the Final

- The final is **Wednesday, December 17, 2014 at 2pm in Harvard Hall 102**.
- We will have a review section to go over these materials on **Sunday, December 14 from 4-6pm in Maxwell Dworkin G125**. 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, there are 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* $T(n) = 3T(n/2) + 19\sqrt{n}$

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

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.*

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?*

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?*

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 $\Sigma = \{a, b\}$)*

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

Write down a formal description of your NFA.

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

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

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.

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

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:*

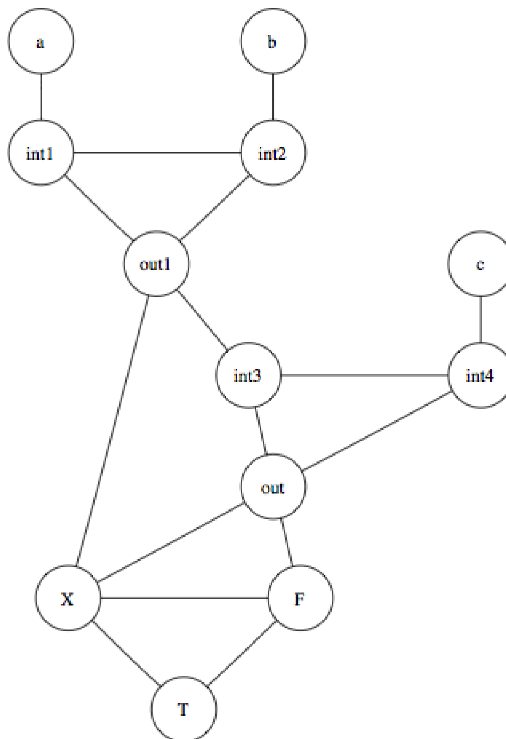
1. *Show that there is a polynomial-time algorithm A_2 for determining whether a Boolean formula has a satisfying truth assignment in which exactly 2 of the Boolean variables are true.*
2. *Show that there is a similar polynomial-time algorithm A_k for any $k \geq 0$, which determines whether a formula 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 $A_0, A_1, \ldots, A_n$ to F and accept iff one of the A_i accepts."*

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?

Exercise. *Consider the following language:*

Subgraph isomorphism : $\{\langle G, H\rangle : G$ *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.*

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.*

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 and that $\overline{A_{TM}}, \overline{A_{WR}}$ are unrecognizable.

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

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

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

Define

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

Is L_1 decidable? Is L_1 in P?

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?

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*$\}$	■				

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*
$\{\langle D_1, D_2\rangle : L(D_1) \subseteq L(D_2)\}$					
$\{\langle M_1, M_2\rangle : L(M_1) \subseteq L(M_2)\}$		■	■		
$\{\langle M, D\rangle : L(M) \subseteq L(D)\}$					

Exercise. *Suppose that aliens land on Earth, presenting humans with a black box B that solves* $\textsc{Halt}_{TM}$.

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.*

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?*

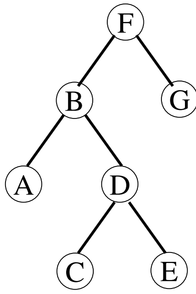
Exercise. *Suppose we are building a binary search tree on a set of data. Normally we would think to make the tree balanced to minimize the search time. But this is not best if we know something about the probability that a node is accessed.*

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?

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. $A = (a_1, a_2, \ldots, a_n)$ where there exists a k such that $a_k < a_{k+1} < \ldots < a_n < a_1 < \ldots < a_{k-1}$, but you don't know what k is. Give an efficient algorithm to find the largest number in this array.*

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.*

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?*

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.*

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.*

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.*

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.*

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?*

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* $1/2^n$*?*

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).*

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.*

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?*

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.

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.*

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.*

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.

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:*

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

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.*

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, $P = \{p_1, \ldots, p_n\}$. 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 $O(nk)$.

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

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.)*