Contents

1 Notes about the Final

- The final is **Friday, May 15, 2015** at **9am** in **Science Center C**.
- We will have a livestreamed and videotaped review section to go over these materials on **Wednesday, May 13** at **7:40-9:40pm** in **Maxwell-Dworkin G-125**. We won't go through all of this material during the review section, so bring 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.

2 Study Tips

(This is the more or less the same as the post at `https://piazza.com/class/i53uki9459o1fn?cid=1162`.)

In general, as you've seen on the midterms, you can probably break down the types of questions you'll be asked into two broad categories: those that test you on more on specific algorithms we've covered, and those that test you on your ability to apply more general algorithmic tools/paradigms.

When studying specific algorithms:

- Understand the main idea behind the algorithm, the runtime, and the space complexity for any algorithm that we've covered.
  - Trying to boil down what the algorithm is doing to a 1-3 sentence summary can help to ensure you understand the key point of the algorithm.
  - For space and time complexity, you want to be able to identify what the most expensive operations are or what the recursion leading to the complexity is.
- Understand the constraints for what types of problems that algorithm solves.
  - In a similar vein, you want to practice applying algorithms to problems, and understanding what the key features are of a problem solved by an algorithm helps you figure out which algorithm to use.
  - For example, if we have an algorithm on DAGs, do you know why it wouldn't work on graphs with cycles? (Try thinking about what would happen to the algorithm if you invalidated one of the assumptions – what step(s) of the analysis would go wrong in this case? Could you adjust the algorithm to fix them?)
- Focus on the big ideas of the analysis of the algorithms above all the details.
  - A good example is FFT; you'll probably get more utility from understanding what FFT does (it lets you multiply two degree-N polynomials in time $O(N \log N)$) and how to use it, than to memorize the algebra that goes on behind the scenes.
- Try thinking about variations of the algorithms. For example, if you have a major step in the algorithm, try thinking about what would happen if tweaked that step. Alternatively, if the algorithm uses one data structure, what would happen if you replaced it with a different one?

When studying more general tools/paradigms:

- Practice applying them.
  - Apart from the problems we've released, you can find others in the textbook and online.
  - One thing that's helpful here is to also solve problems where you don't know what tool you're supposed to be using, so that you can practice choosing them. We've uploaded a set of unlabeled practice problems (see the Announcements tab on Canvas) to help with this.
- Try to break down the process into pieces (e.g. the three steps of dynamic programming, the steps for showing NP-completeness).
- Understand what the key characteristics are that make a tool work well.
  - For example, dynamic programming tends to work better when you can break your problem down into slightly smaller subproblems (say one size smaller), whereas divide and conquer works when you can break your problem down into significantly smaller problems (say half the size).
- When you're working on the problems, a lot of times the most difficult piece is not the details of analysis/proof, but the initial choices you make in set.
  - For dynamic programming and divide-and-conquer, the first step is to choose a subproblem. However, a lot of times the difficulty in getting to the answer is not in the analysis after this point, but in choosing the subproblem itself (so if you find yourself getting stuck, try using a different subproblem).
  - For dealing with P/NP, when doing reductions you generally want to choose a problem that's as close to the original as possible; choosing a subproblem that's further away will probably force you to do a lot of extra work when you're trying to prove the reduction.

3 Mathematics

3.1 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.

3.2 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, i.e. $\sum_{s \in S} s \cdot \mathbb{P}(X = s)$.

A few useful facts:

- 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, $\mathbb{E}(X) = \displaystyle\sum_{k=0}^{\infty} \mathbb{P}(X > k)$.

Two useful tools:

- **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 (Coupon Collector)**.** *There are* n *coupons* $C_1, \ldots, C_n$*. Every time you buy a magazine, a uniformly random coupon* C_i *is inside. How many magazines do you have to buy, in expectation,*

to get all n coupons? How many magazines do you have to buy to get all n coupons with a failure probability of at most P?

Solution.
a) You need $\Theta(n \log n)$ to get all n coupons.

b) To get all n coupons with probability P, it's enough to buy $\Theta(n \log n \log \frac{1}{P})$.

Proofs:

a) Once you've collected k coupons, buying a new magazine gives you a new coupon with probability $1 - \frac{k}{n} = \frac{n-k}{n}$. Thus you need to flip $\frac{n}{n-k}$ coupons to get the $(k+1)$-st one. Thus the expected number of magazines you have to buy is $\sum_k \frac{n}{n-k} = n \cdot \sum_{t=1}^{n} \frac{1}{t}$ (substitute $t = n - k$). This is the n-th Harmonic number, which we can check is $\Theta(n \log n)$ by an integral.

b) By Markov's inequality, the probability you don't get all coupons after $2 \cdot$ (expected number of steps) is at most $\frac{1}{2}$. Thus the probability this happens $\log \frac{1}{P}$ times in a row is $2^{-\log \frac{1}{P}} = P$.

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 $\log n$).

4 Deterministic Data Structures

4.1 Heap

A **Heap** is a data structure to enable fast deletion of the maximal or minimal element in a dynamic list. Consequently, we often use them to implement priority queues.

The following are operations and runtimes for a binary Max-Heap:

- Build-Heap: Turn an (unordered) list of elements into a heap in time $O(n)$
- Max: Identify the maximal element in time $O(1)$
- Remove-Max: Remove the maximal element in time $O(\log n)$
- Insert: Insert a new element in time $O(\log n)$

While we often think of heaps as a tree, note that they can also be stored in memory as an array.

Exercise. *Suppose that we implement* Remove-Max *as follows:*

- *Remove the maximal element*
- *Take the larger of its two children, and move that one up into the space previously occupied by the maximal element. Now repeat this recursively for the element just moved, until we reach the leaves of the tree.*

What problem(s) might this implementation run into?

Solution.
This method does not guarantee that the max-heap will remain balanced, consequently, it could blow up the runtimes of future operations. With a sufficiently unlucky series of operations, this could lead to the heap effectively being an ordered list instead of a balanced binary tree.

4.2 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.

4.3 LCA and RMQ

LCA: given two nodes in a tree, what is their last common ancestor?

RMQ: for an array of numbers, what is the index of the smallest number in a given subarray?

We can solve both using linear processing time, linear memory, and constant query time.

1. Linear-time reduction from LCA to RMQ:
   (a) DFS array V: an array of size $2n-1$ that keeps track of nodes visited by DFS.
   (b) Level array L: same length as V, keeps track of distance from root at each node.
   (c) Representative array R: $R[i]$ is the first index of V containing the value i.
   (d) LCA(u, v) is now same as RMQ$(R[u], R[v])$ on L.
2. The reduction above guarantees that adjacent elements of L differ by no more than one (± 1 RMQ). We can take advantage of this to get a linear-time preprocessing/memory algorithm for LCA and RMQ.
3. It is also possible to reduce general RMQ to ± 1 **RMQ**] in linear time.
4. Consequently, we can solve these in $O(n)$ space and preprocessing time, and $O(1)$ query time.

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.

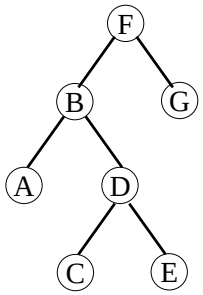
Exercise. *Suppose we are building a binary search tree on a set of data. Normally we would think to make the tree balanced to minimize the search time. But this is not best if we know something about the probability that a node is accessed.*

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 search tree with B as the root, A and F its children, D, G the children of F, and C,E the children of D. Then, the average number of comparisons needed is 2.1, 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, fast integer multiplication, and median finding. 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.*

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 $M[i, j] = 0$, 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. Additionally, it is possible to do so in constant space because $B[i]$ and $S[i]$ depend only on $B[i-1]$ and $S[i-1]$.

5.4 FFT

Recall that FFT allows us to find the product of two n-degree polynomials in time $O(n \log n)$, rather than the $O(n^2)$ given by the naive approach (with some assumptions of floating point accuracy).

In class, we saw an example of FFT for string matching; that is, let $P = p_0 p_1 \dots p_{m-1}$ and $T = t_0 t_1 \dots t_{n-1}$ be two binary strings. We want to find all occurrences of P in T. To recap, the algorithm is as follows:

- Obtain $P' = a_0 a_1 \dots a_{2m-1}$ and $T' = b_0 b_1 \dots b_{2n-1}$ by replacing each 0 with 01 and each 1 with 10.
- Multiply the two polynomials $A(x) = a_{2m-1} + a_{2m-2}x + \ldots + a_0 x^{2m-1}$ and $B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_{2n-1}x^{2n-1}$. Using the FFT, this can be done in time $O(n \log n)$.
- Now, examine each coefficient of the polynomial $A(x) \cdot B(x)$; the number of coefficients equal to m will be the number of times P occurs in T.

Exercise. *Suppose you are given two fair n-sided dice, where each face of the dice contains some number in $\{1, 2, \dots, k\}$. Given an efficient algorithm for determining the probability distribution of the sum of the two dice (you may assume infinite precision arithmetic in constant time).*

Solution.
For each die, create a polynomial $\frac{1}{n}\left(a_1 x^1 + \ldots + a_k x^k\right)$, where a_i is the number of faces on which the number i appears. Then, use FFT to multiply the two polynomials; the coefficient of x^s in the result is the probability that the two dice will sum to s.

This works because each polynomial represents the set of probabilities for the rolls of that die. Then, when we multiply them, the probabilities are multiplied while the exponents (i.e. the values of the rolls) are added.

This takes time $O(n + k \log k)$ due to the time to construct the polynomials and the FFT.

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 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) or working on it much easier.

We've seen two major types of graph traversals:

- **Depth-First Search (DFS)**: keep moving along edges of the graph until the only vertices you can reach have already been visited, and then start backtracking. This also leads to a **topological sort** of a directed acyclic graph, which orders the vertices by an ancestor-descendant relation. Runtime: $O(|V| + |E|)$.
- **Breadth-First Search (BFS)**: visit the vertices in the order of how close they are to the starting vertex. Runtime: $O(|V| + |E|)$ if all the edges have length 1, $O(|E| \log |V|)$ for Djikstra's algorithm on nonnegative edge lengths with a binary heap.

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.

6.2 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 reasonable 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.

6.3 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.

Exercise. *Suppose that we are given a graph with the following guarantee: for any two disjoint sets S, T of vertices, there exists some edge between a vertex in S and one in T with length no more than $1/\max(|S|, |T|)$. Find a bound on the size of the minimum spanning tree for this graph, in terms of the number of vertices.*

Solution (Solution thanks to Lily Tsai.)**.**
We create the tree similarly to Prim's algorithm.

In particular, start with sets of vertices of size 1 and $n-1$; we know that there exists an edge of size no more than $1/(n-1)$ from the single vertex to one of the other vertices. Add that edge. Now, we have sets of size 2 (connected) and $n-2$ (disconnected), and can find an edge of size no more than $1/(n-2)$ from one of the two vertices into a disconnected one. We can continue this process until we have one connected set of size $\lceil n/2 \rceil$ (the tree so far) and one entirely disconnected set of $\lfloor n/2 \rfloor$ vertices.

This is the point at which we switch from using the bound 1/(#disconnected vertices) for a bound on the edge length to 1/(#vertices in tree so far).

Now there is some disconnected vertex of distance at most $1/\lceil n/2 \rceil$ from our tree so far, then one of $1/\lceil n/2+1 \rceil$, etc. Finishing out this process gives a total edge length of

$$2 \sum_{i=\lceil n/2 \rceil}^{n-1} \frac{1}{i} \leq \frac{2}{n-1} + 2 \int_{i=\lceil n/2-1 \rceil}^{n-2} \cdot \frac{1}{i} di = \frac{2}{n-1} + 2 \ln 2 \approx 2 \ln 2.$$

Exercise (Previous problem continued)**.** *What if, in the previous problem, we had $1/\min(|S|, |T|)$ instead of $1/\max(|S|, |T|)$?*

Solution.
Now, we use an idea that more ressembles Kruskal's algorithm.

Let n be the number of vertices; assume that n is a power of 2. Suppose that at some point in constructing the MST, we have $n/2^i$ connected sets of size 2^i. Then, we can connect pairs of them into $n/2^{i+1}$ sets of size 2^{i+1} with the addition of $n/2^i$ edges of size at most $1/2^i$.

Summing over all i, we get a total size of at most

$$\sum_{i=0}^{\log_2 n-1} \frac{n}{2^i} \cdot \frac{1}{2^i} = n \sum_{i=0}^{\log_2 n-1} \frac{1}{4^i} \leq \frac{4n}{3}.$$

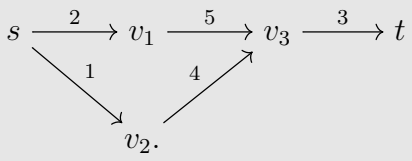
6.4 Network Flows

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. **Karger's algorithm** To try to find a minimum cut on a graph where all edges have weight 1, we randomly contract edges until there are only two vertices left, and then output the weight of the cut which separates the two vertices.
   (a) To **contract** an edge (u, v) means replacing (u, v) by a new node w, and then replacing u or v with w in all edges of the graph where they occur.
   (b) With probability at least $1/\binom{n}{2}$, Karger's algorithm will output a particular minimum cut.
4. 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



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

Exercise. *Suppose you are in charge of the electricity company and given a graph of transmission line capacities, want to determine whether it is possible to send h_i watts of energy simultaneously to the ith house, for each i. Give an efficient algorithm for doing so, assuming that all the energy comes from a single power station.*

Solution.
Create a sink node, and for each house, add an edge of capacity h_i from that house to the sink. Then, it is clear that there is a way to simultaneously send enough energy to all the houses if and only if the maximum flow in this graph is $\sum_i h_i$ (this is the same idea that was used in one of the homework problems).

This runs in the same time as our max flow algorithms.

Exercise. *Now, suppose that h_i is the amount of electricity the ith house needs during the day, while it instead needs $g_i < h_i$ electricity during the night. However, you would like to minimize the drop in electricity that flows through each transformer (i.e. intermediate vertex) between the day and night. Give a polynomial-time algorithm to determine the smallest maximum difference that can achieved in the network.*

Solution.
We can phrase this as a linear program. First, we set up the two linear programs for the flow during the day and night, for the graphs constructed in the previous problem, i.e. let d_e be the flow through an edge e during the day, n_e during the night, and c_e the edge's capacity. Then, these problems have constraints:

$$\begin{aligned}
n_e, d_e &\geq 0 \\
n_e, d_e &\leq c_e \\
\sum_{(u,v)} d_{(v,w)} &= \sum_{(v,w)} d_{(v,w)} \forall v \text{ not a source or sink} \\
\sum_{(u,v)} n_{(v,w)} &= \sum_{(v,w)} n_{(v,w)} \forall v \text{ not a source or sink} \\
\sum_{(h,t)} d_{(h,t)} &= \sum_i h_i \text{ for } t \text{ the sink} \\
\sum_{(h,t)} n_{(h,t)} &= \sum_i g_i \text{ for } t \text{ the sink.}
\end{aligned}$$

In particular, the last two constraints guarantee that the desired flows are achieved during each the day and night. Now, let z be the variable we want to minimize. Then, we have

$$z \geq \sum_{u,v} d_{u,v} - \sum_{u,v} n_{u,v} \forall v \text{ not a source, house, or sink .}$$

6.5 2-player Games

An equilibrium in a 2-player game occurs when neither player would change his strategy even knowing the other player's strategy. This leads to two types of strategies:

- **Pure strategy**: a player always chooses the same option
- **Mixed strategy**: both players randomly choose an option using some probability distribution

For a player, option A is **dominated** by option B if, no matter what the other play chooses, the first player would prefer B to A. Because players will never play dominated strategies, the first thing to do when solving a 2-player game by hand is to eliminate any that you find.

Exercise. *True or false: a 2-player game may only have a single pure strategy for each player.*

Solution.
False; consider the following game, in which either both players could play their first option or both could play their second:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

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 Algorithms

In general, we have two kinds of probabilistic algorithms:

- Las Vegas: always correct, but the worst-case bound for the running time may be bad (so instead we bound the expected running time)
- Monte Carlo: is correct with good probability (but not always), but we can give a good bound for the worst-case running time

In general, we have tradeoffs between runtime and accuracy:

- In many cases, we can turn a Las Vegas algorithm into a Monte Carlo one by stopping it after time $cE[\text{runtime}]$, for some constant c – by Markov's inequality, the failure rate is no more than $1/c$.
- We can reduce the failure probability of a Monte Carlo algorithm by running it multiple times.

7.1 Hashing

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

Hashing-based data structures (e.g. **hash tables**) 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).

In reality, it is hard to get perfect randomness, where all elements hash independently. Consequently, we instead think about families of hash functions. Such a family $\mathcal{H}$ of hash functions is **universal** if, for all $x \neq y$ in $\{0, \ldots, n-1\}$ and for an h chosen uniformly for $\mathcal{H}$,

$$\Pr(h(x) = h(y)) \leq \frac{1}{m}.$$

Exercise. *Suppose you have a hash table with a perfectly random hash function. Write down an expression for the expected number of elements you can insert until there is a collision.*

Solution.
Let X be the random variable for the expected number of elements. Then, we know that

$$E[X] = \sum_{i=1}^{n} \Pr[X \geq i].$$

However, we know that $\Pr[X \geq i]$ is the probability that the first i elements are hashed to distinct values, i.e. $\frac{m}{m} \cdot \frac{m-1}{m} \cdot \ldots \cdot \frac{m-i+1}{m}$. Consequently,

$$E[X] = \sum_{i=1}^{m} \frac{m}{m} \cdot \frac{m-1}{m} \cdot \ldots \cdot \frac{m-i+1}{m}.$$

Exercise. *Does your bound in the previous exercise still hold for every set of elements if the hash function is instead selected from a universal hash family?*

Solution.
No. In fact, now, consider the probability that, after i elements $x_1, \ldots, x_i$ have been inserted, there are no collisions. The best bound on $\Pr[X \geq i]$ that we are guaranteed, by a union bound, is

$$\Pr[X \geq i] = 1 - \Pr[\text{exists a collision}] \geq 1 - \sum_{1 \leq j < k \leq i} \Pr[h(j) = h(k)] \geq 1 - \frac{i(i-1)}{2m}.$$

Exercise. *Now suppose instead that we use two independent perfectly random hash functions, and that we check the results of both hash functions to try to find a place to insert an element. Write down an expression for the expected number of elements inserted before one comes along for which both options are already full.*

Solution.
Now, assuming that $i-1$ elements have already been placed, the probability that both slots of the next element are occupied is $\left(\frac{i-1}{m}\right)^2$, so $\Pr[X \geq i] = \left(1 - \left(\frac{0}{m}\right)^2\right) \cdot \left(1 - \left(\frac{1}{m}\right)^2\right) \cdot \ldots \cdot \left(1 - \left(\frac{i-1}{m}\right)^2\right)$. Consequently,

$$E[X] = \sum_{i=1}^{m} \left(1 - \left(\frac{0}{m}\right)^2\right) \cdot \left(1 - \left(\frac{1}{m}\right)^2\right) \cdot \ldots \cdot \left(1 - \left(\frac{i-1}{m}\right)^2\right).$$

7.2 Skip Lists

A **skip list** is a sorted randomized data structure with expected space $O(n)$ and expected runtime of $O(\log n)$ for predecessor queries, insertions, or deletions. It consists of a layered series of lists, each of which contains, on expectation, half of the elements in the previous list. To find an element's location, we start with the smallest list and work our way down, finding the relative location of that element in each subsequent list.

Exercise. *Describe how you would modify a skip list to identify, in expected $O(\log n)$ time, the position of an element in the list.*

Solution.
This is the same modification as is needed to be able to look up the ith smallest element in logarithmic time: have each node track how many items it has between it and the previous node in its level at the bottom list.

Then, you can add or subtract these differences while searching for an element by walking through the lists.

7.3 Quicksort and Quickselect

In **Quicksort**, we randomly choose a pivot element, separate the list into elements larger and smaller than that element, and recursively sort those two sublists. The expected runtime is $O(n \log n)$.

In **Quickselect**, we randomly find the kth smallest element of a list as follows: randomly choose a pivot element and separate the list into elements larger and smaller than that element. Figure out which sublist the kth smallest element should fall into (and what position it will be in), and then recursively search for it in that sublist. The expected runtime is $O(n)$.

Exercise. *Suppose that in Quicksort, instead of choosing one pivot each time, we randomly choose two pivots each time, divide the list into three pieces based on these pivots, and recurse. Determine the expected runtime of this algorithm.*

Solution.
The expected runtime is still $O(n \log n)$. Intuitively, note that even were we to choose pivots each time that divided the list exactly in three, then our recurrence relation would be $T(n) = 3T(n/3) + O(n)$, which is $O(n \log n)$ by the master theorem.

Meanwhile, looking at the specific analysis with randomization, note that the same argument still holds as for regular Quicksort: two elements i, j are compared if and only if i or j is chosen as a pivot before any element between them is. But, since we choose two pivots at a time, the probability that i or j is one of them is at most $\frac{4}{j-i+1}$ instead of the $\frac{2}{j-i+1}$ that we had originally, which at most changes the expected runtime by a constant.

7.4 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 $O(n^2)$.

Exercise. *Suppose we start at the point $n-1$, and at each step increase by 1 with probability $2/3$, decrease by 1 with probability $1/3$. What is the probability we reach n before 0?*

Solution.
Let $T(i)$ be the probability of reaching n before 0 starting at point i. Then,

$$\begin{aligned} T(n) &= 1 \\ T(i) &= \frac{T(i-1)}{3} + \frac{2T(i+1)}{3}, n > i > 0 \\ T(0) &= 0. \end{aligned}$$

To solve this, notice that the middle equation can be rearranged as $2(T(i+1)-T(i)) = T(i)-T(i-1)$. Meanwhile, the first and last equations give

$$1 = T(n) - T(0) = \sum_{i=1}^{n}(T(i) - T(i-1)) = \sum_{i=1}^{n} 2^{n-i}(T(n) - T(n-1)) = (2^n - 1)(T(n) - T(n-1)).$$

Solving, we get $T(n-1) = 1 - \frac{1}{2^n-1}$.

8 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. *Consider the following greedy algorithm for* Max 3-Sat*, given an instance where each clause involves three distinct variables: at each point in time, assign a value to one more variable such that the number of clauses satisfied so far is* increased *by the largest number possible.*

Prove that this gives a 3/4*-approximation, and not any better.*

Solution.
First, we claim that this necessarily satisfies 3/4ths of the clauses, and hence must be a 3/4-approximation. The main idea is that for each clause that is not satisfied under the greedy algorithm, because it involves 3 distinct variables, we must have chosen at 3 points in time a variable assignment that did not satisfy it. However, because at each point in time the choice must have satisfied more clauses than ones which would have been satisfied with the opposite assignment, this means that each unsatisfied clause must correspond to three satisfied one.

More concretely, suppose that on the ith assignment, it increased the number of satisfied clauses by s_i, and that there were u_i clauses not satisfied so far that would have had we made the opposite at that point in time. Because we are making assignments greedily, it must be true that $s_i \geq u_i$. The total number of satisfied assignments is $\sum_i s_i$. Each unsatisfied clause must have contributed to the u_i terms for three distinct i's, so the number of unsatisfied clauses is at most $\sum_i u_i/3$. Consequently, we have that (# satisfied clauses) ≥ 3(# unsatisfied clauses), giving a 3/4 approximation factor.

Secondly, we give an example which proves that the approximation factor can become arbitrarily close to 3/4. Suppose we have k copies of the clause $(\neg x_1) \wedge (\neg x_2) \wedge (\neg x_3)$. Then, suppose we have $k+1$ clauses of the form $x_i \wedge x_\alpha \wedge x_\beta$, for each of $i = 1, 2, 3$, with α and β distinct in each clause. Clearly, it is possible to satisfy all the clauses by setting x_1 to be false and all the other variables to be true. However, the greedy algorithm will set each of x_1, x_2, x_3 to be false, which only leads to $\frac{3k+3}{4k+3}$ satisfied clauses. As $k \to \infty$, this becomes arbitrarily close to 3/4.

Exercise (Challenge)**.** *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 $O(nk)$.

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 (Challenge)**.**
Consider an optimal choice of $S^* = \{s_1, \ldots, s_k\}$, 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 (Challenge)**.** *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.

9 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.

   (c) *When doing a reduction, remember to check that it's going in the right direction!*

   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

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

   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 a 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.

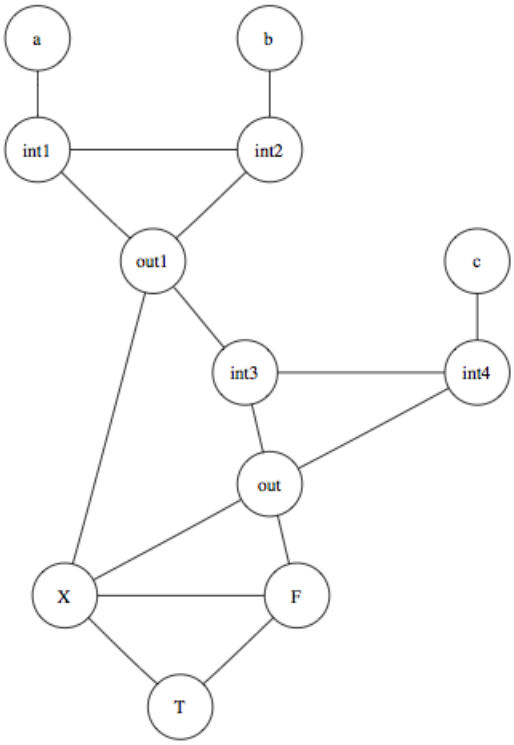
Exercise. *Here is another NP-complete problem:*

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:*

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

(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.