## Appendix K

# Scrambled problems

The exercises can be more fun when you don't know which chapter they come from. Here are all the exercises from Chapters 1–14 and 16–19, in random order.

**Exercise K.1.** Let $k$ be fixed. Suppose you had access to a polynomial time algorithm for the decision version of $k$-coloring. Use this algorithm to obtain a polynomial time algorithm that obtains a $k$-coloring of a graph (if one exists).

**Exercise K.2.** In this exercise, we develop a refined analysis that can reduce the additive error substantially in (arguably realistic) settings when the stream is dominated by heavy hitters.

Let $S$ denote the sum of frequency counts of all elements that are *not* $\epsilon$-heavy hitters:

$$S = \sum_{e:p_e<\epsilon} f_e.$$

Note that $S \leq m$, and $S$ might be much less than $m$ when the stream is dominated by heavy hitters.

Show that, by increasing the parameter $w$ (in the `count-min` data structure) by a constant factor, and still using universal hash functions, one can estimate the frequency of every element with additive error at most $\epsilon S$ with high probability in $O(\log(n)/\epsilon)$ space.[1]

**Exercise K.3.** Recall that in chess, a *rook* (♖) is a piece that can move horizontally or vertically as far as it wants. (It cannot move diagonally.) We say that a rook on a particular square is *attacking* another chess piece at another square if the rook can be moved horizontally or vertically from its square to the square of that chess piece.

[1] *Hint*: It might be helpful to think about the special case of $S = 0$.

Consider the following problem which we call the "$n$-rooks problem". At a high level, there is an $n \times n$ board where some of the squares have been marked off as "unavailable". The remaining squares are "available". The goal is to place $n$ rooks in the available squares so that none of the rooks are attacking each other. We call this a "non-attacking placement" of the $n$ rooks. (While rooks cannot be placed on unavailable squares, they can still move over unavailable squares to attack other pieces on the other side of unavailable squares.)

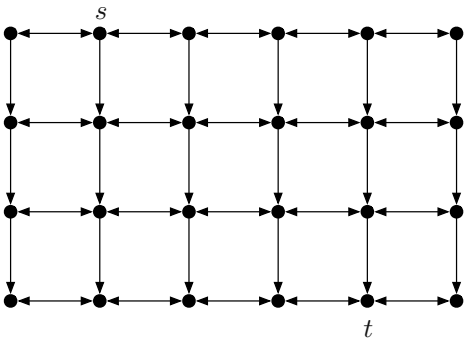
Below we draw a non-attacking placement of 4 rooks on a $4 \times 4$ chess board. Unavailable squares are marked by `X`. Rooks are marked by ♖.

| | | | |
|---|---|---|---|
| X | ♖ | | X |
| ♖ | X | X | X |
| | X | ♖ | |
| X | | X | ♖ |

Formally, the input consists of an $n \times n$ bit-array $A[1..n, 1..n] \in \{\texttt{true}, \texttt{false}\}^{n \times n}$, where $A[i, j] = \texttt{true}$ indicates that square $(i, j)$ is available, and $A[i, j] = \texttt{false}$ indicates that square $(i, j)$ is unavailable. The goal is to decide if there is a non-attacking placement of $n$ rooks in the available squares. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.4.** Suppose you only have access to a coin that flips heads with a known probability $p$, and tails with (remaining) probability $1 - p$. Describe and analyze a protocol that uses a limited number of tosses of this biased coin in expectation (the smaller the better) to simulate 1 coin toss of a fair coin. (The expected number of biased coin tosses you make may depend on $p$.)

**Exercise K.5.** Let $G = (V, E)$ be a directed graph with real-valued (and possibly negative) edge lengths $\ell : E \to \mathbb{R}$, and $s, t \in V$. We are interest in the length of the shortest $(s, t)$-path for a special class of "downwards grid graphs" $G$.



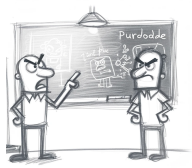

A "downward grid graph" $G$ is parametrized by a height $m \in \mathbb{N}$ and a width $n \in \mathbb{N}$. All the vertices are arranged in an $m \times n$ grid. We let $(i, j)$ denote the vertex in row $i$ and column $j$. There are edges between every adjacent pair of edges except going up. More specifically, for $i \in [m]$ and $j \in [n]$, there is an arc

- (Down) to $(i + 1, j)$ when $i < m$;
- (Right) to $(i, j + 1)$ when $j < n$; and
- (Left) to $(i, j - 1)$, when $j > 1$.

In particular, there are no edges going "up" from $(i, j)$ to $(i - 1, j)$. Note that $G$ has $O(mn)$ vertices and $O(mn)$ edges.

Consider the problem of computing the length of the shortest $(s, t)$-path in a downward grid graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.6.** As CEO of the fast growing tech company Purdoodle, you've decided to put together an internal conference on various topics in the field. There are $m$ different topics $T_1, \ldots, T_m$, and $n$ employees $E_1, \ldots, E_n$. Each employee has filled out a survey where they each select (exactly) 2 topics they are interested in and your goal is to assign each of them to a topic.




However, some pairs of employees are known to have had heated arguments and conflicts over particular topics. In these cases, you don't want to assign both of these employees to that same troublesome topic. The high-level goal is to assign employees to topics as to minimize, or even eliminate, these conflicts.

For the sake of concreteness, you can assume your input consists of:

- The list of survey responses of the form $(E_j, T_h, T_i)$, meaning that employee $E_j$ is interested in topics $T_h$ and $T_i$. (Each $E_j$ appears in one entry, and for each $(E_j, T_h, T_i)$, $T_h$ and $T_i$ are different.)
- A list of $p$ conflicts of the form $(E_j, E_k, T_i)$, meaning that $E_j$ and $E_k$ have previously conflicted over topic $T_i$, so you don't want to assign both of them to topic $T_i$. You can assume that $T_i$ was listed as a topic of interest by both $E_j$ and $E_k$. (It is easy to filter out other conflicts in a preprocessing step.)

For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

1. Suppose you want to have zero conflicts. The goal is to assign each employee to 1 of their 2 chosen topics such that there are no conflicts in any of the topic sessions, or declare that it is impossible to schedule the conference without any conflict.

2. (Harder) If having no conflicts is unrealistic, then we can at least try to minimize the number of conflicts. The problem is to assign each employee to 1 of their 2 chosen topics as to minimize the total number of conflicts.

**Exercise K.7.** Let $h : [n] \to [\ell]$ be an ideal hash function, with $\ell \geq n$. What is the *exact probability* that $h$ has no collisions (i.e., $h$ is injective)?

**Exercise K.8.** Let $G = (V, E)$ be a directed graph with positive edge weights, and $s, t \in V$. Recall that Dijkstra's algorithm returns the lengths of the shortest paths from $s$ (as well as the shortest paths themselves by incorporating parent pointers). However there may be many shortest $(s, t)$-paths for a particular $t$. Suppose we want to find the shortest (weighted) $(s, t)$-paths with the fewest number of edges (to break ties). Consider the problem of computing both the shortest path lengths as well as the minimum number of edges in each shortest path. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.[2]

**Exercise K.9.** Consider the following variation of SAT, which we will call *opposite-SAT*. The input consists of a collection of $m$ disjunction clauses over $n$ variables $x_1, \ldots, x_n$. The goal is to decide if there is an assignment such that every disjunction evaluates to `false`. For this problem, either (a) design and analyze a polynomial time algorithm, or (b) show that a polynomial time algorithm implies a polynomial time algorithm for SAT.

**Exercise K.10.** Let $G = (V, E)$ be a directed graph, and consider the problem of adding edges to $G$ to make $G$ strongly connected. The goal is to make $G$ strongly connected by adding the minimum number of edges. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

[2]Technically, there could be as many as $n!$ paths in which case an arithmetic operation would take $O(\log(n!)) = O(n \log n)$ time. For this exercise, let us assume for simplicity that arithmetic operations take $O(1)$ time anyway.

**Exercise K.11.** Suppose you are given a set of $n$ points $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ in the plane.

1. Design and analyze an algorithm computing the maximum cardinality $|S|$ of any subset $S \subseteq P$ where every pair of points in $S$ is comparable. (Cf. exercise K.107 on page 815.)

2. Design and analyze an algorithm computing the maximum cardinality $|S|$ of any subset $S \subseteq P$ where every pair of points in $S$ is incomparable.

**Exercise K.12.** Suppose you had an $(s, t)$-flow $f$. We know that there exists an $(s, t)$-path packing of the same size as $f$; here we are interested in algorithms that take $f$ and compute such a path packing. Such a path packing is called a *flow decomposition* of $f$.

Design and analyze an algorithm that, in $O(m^2)$ time, computes a maximum path packing $x$ of the same size as $f$, such that:

1. There are at most $m$ distinct paths (with nonzero value) in $x$.

2. If $f$ is integral, then $x$ is also integral.

**Exercise K.13.** Let $A[1..n]$ be an array of elements. The *frequency* of an element is the number of times it appears in $A[1..n]$. An element is a *majority element* if it has frequency at least $n/2$.

If the elements are comparable then we can identify the majority element by sorting all the elements and scanning the sorted list. Here we do *not* assume that the elements are pairwise comparable; we can only test if two elements $A[i]$ and $A[j]$ are equal.

1. Write a recursive spec for the majority element problem.

2. Design a divide-and-conquer algorithm (implementing your recursive spec) for the majority element problem that divides the input in half, and runs in $O(n \log n)$.[3]

3. Prove your algorithm is correct by induction, where the recursive spec is your induction hypothesis.

4. Analyze the running time of your algorithm.

[3]There is also a well-known $O(n)$ time algorithm which we are not asking for.

**Exercise K.14.** Karatsuba's algorithm gave an $O(n^{.585})$ time algorithm for multiplying two $n$-bit integers. Design and analyze a nearly linear time algorithm (i.e., $O\left(n \log^{O(1)} n\right)$ time) for integer multiplication.

**Exercise K.15.** Recall the recursive algorithm for longest path in DAG's.

1. Describe a DAG with $n$ vertices where the recursive algorithm (without dynamic programming) would take $O(n!)$ time.
2. More generally, show that for any integers $k \leq n$, there is a DAG with $n$ vertices such that
   (a) Every vertex $v$ has out-degree $\leq k$.
   (b) The recursive algorithm would take $(n-k)^k(k-1)!$ on this input.

**Exercise K.16.** Recall that in foreign exchange, different currencies can be exchanged at various exchange rates. For example, at the time of writing, one can exchange 1 US dollar for about 0.89 Euro's. *Currency arbitrage* occurs when you start with some quantity of one currency, and make a sequence of exchanges through different currencies and end up with even more of the original currency than you started with.

Suppose we have $n$ countries and for every (ordered) pair of countries $X$ and $Y$ we have an exchange rate $r_{X,Y}$; meaning, one unit of currency $X$ can be exchanged for $r_{X,Y}$ units of currency $Y$. Consider the problem of identifying currency arbitrage starting from a fixed currency $X$, or deciding that no arbitrage is possible. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.17.** This problem models the following scenario. To prepare for the final, you and your classmate have decided to try to solve all $n$ of the exercises in the textbook. Due to time constraints, you've decided to split up the problems. To try to make the total effort even, each of you went through and skimmed all the problems, and for each problem, estimated the number of minutes it would take to solve the problem. (Thus producing two lists of estimates, one for each person.) The goal is to assign each exercise to you or your classmate so as to minimize the maximum time that either of you would spend solving their assigned problems. We call this problem the "optimum study allocation problem".

Formally, we let $\{1, \ldots, n\}$ represent the $n$ exercises. The input consists of two lists of natural numbers $A[1..n]$ and $B[1..n]$. For $i = 1, \ldots, n$, $A[i]$ represents your

estimated time for problem $i$ and $B[i]$ represents your classmate's estimated time for problem $i$. The goal is to divide $\{1, \ldots, n\}$ into two sets, $S \subseteq \{1, \ldots, n\}$ and $T = \{1, \ldots, n\} \setminus S$, to minimize the maximum of

$$\sum_{i \in S} A[i] \text{ and } \sum_{j \in T} B[j].$$

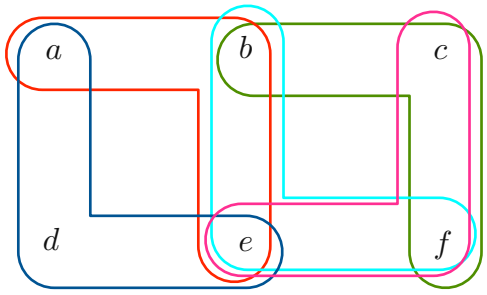
Here $\sum_{i \in S} A[i]$ represents the total time you estimate to solve all the problems in $S$ and $\sum_{j \in T} B[j]$ represents the total estimated time for your partner to solve all the problems in $T$.

For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.18.** Recall that a directed graph $G = (V, E)$ consists of a set of vertices $V$ and edges $E$ that are ordered pairs of vertices $(u, v)$ (where $u, v \in V$). A walk in $G$ is a sequence $v_1, v_2, \ldots, v_k$ where each consecutive pair, $(v_i, v_{i+1})$, is an edge.

Generalizing this idea, we define a *rank-3 directed hypergraph* $G = (V, E)$ as consisting of a set of vertices $v$, and "directed hyperedges" $E$ which consist of *ordered* triples $(a, b, c)$ where $a, b, c \in V$. ($a, b, c$ don't have to be distinct.) A *feasible walk* in the hypergraph $G$ is a sequence of vertices $v_1, v_2, \ldots, v_k$ with at least 3 vertices ($k \geq 3$) and such that each consecutive triple, $(v_i, v_{i+1}, v_{i+2})$ for $i = 1, \ldots, k-2$, is a directed hyperedge.

Below is a picture of a directed hypergraph with 5 directed hyperedges, listed on the right. (Note that the drawing does not convey the order of vertices within each edge; this should be inferred from the list of directed hyperedges.)




$(a, b, e)$
$(a, d, e)$
$(b, e, f)$
$(b, c, f)$
$(e, f, c)$

The walk

$$a \to b \to e \to f$$

is a feasible $(a, f)$-walk because each consecutive subsequence of 3 vertices – $(a, b, e)$ and $(e, b, f)$ – are directed hyperedges. The $(a, f)$-walk

$$a \to d \to e \to f$$

is *not* a feasible walk because $(d, e, f)$ is *not* a directed hyperedge. Meanwhile, there is no feasible walk from $d$ to any other vertex because there is no directed hyperedge starting from $d$. (Similarly for $c$ and $f$.)

Design and analyze an algorithm that, given a directed hypergraph $G = (V, E)$, and vertices $s$ and $t$, decides if there is a feasible walk from $s$ to $t$. When analyzing the running time, we let $n = |V|$ and $m = |E|$. You can assume that $E$ is given as a list of triplets.

**Exercise K.19.** Let $G = (V, E)$ be a connected graph, and $F, S \subseteq E$. Decide if the following statements are true or false.

1. $F$ is spanning iff for every cycle $C$ in $G$ with $k$ edges, $F$ contains at least $k - 1$ edges. [4]
2. $S$ is spanning iff $S$ contains a spanning forest.
3. $S$ is spanning iff $|S| \geq n - 1$.
4. If $G$ has distinct edge weights then $G$ has a unique MST.
5. If $F \subseteq \text{span}(S)$, then $|F| \leq |S|$.
6. If $F$ is a forest, and $F \subseteq \text{span}(S)$, then $|F| \leq |S|$.

**Exercise K.20.** A *stacqueué* is a French list-like data structure supporting both the standard operations of a stack and a queue:

1. dequeue: removes an item from the beginning of the list.
2. push (a.k.a. enqueue): inserts an item at the end of the list.
3. pop: removes an item from the end of the list

[4] To be clear, a cycle specifically means a circuit with no repeating vertices (except at the common vertex where the walk starts and ends).

Recall that a stack implements the `push` and `pop` operations in $O(1)$ worst case time. Design and analyze a stacqueué using only stacks that supports each operation in $O(1)$ amortized time.

**Exercise K.21.** Every semester we have to schedule PSO's for all the students, which is complicated by (a) room availability mixed with (b) time availability both with respect to the students and the rooms. Suppose there are $m$ total students, $n$ available time slots, and $p$ different rooms. Suppose we had the following information.

1. For each student (indexed by) $i \in [m]$, there is a subset of times $T_i \subseteq [n]$ of times when they can attend PSO.
2. For each time slot $j \in [n]$, there is a subset of rooms $R_j \subseteq [p]$ available at that time.
3. For each room $k \in [p]$ there is a maximum capacity $C_k$ of the number of students that can fit in the room at any given time.

From this information we have the following scheduling problems.

1. Consider the problem of deciding if there is a way to schedule all the students to times and rooms while respecting all the constraints listed above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.
2. In addition to making sure each room can fit all its students, we also want to minimize the maximum number of students in any room at any time, to decrease the maximum load on the TA's. Consider the problem of creating a schedule that minimizes the maximum number of students in any room, while also respecting the constraints listed above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.22.** Design and analyze a deterministic 3/4-approximation algorithm for max-SAT.[5]

[5]You may first want to design and analyze a deterministic $(1 - 1/e)$-approximation algorithm for max-SAT.

**Exercise K.23. It's-a me, Mario!** This problem is inspired by Super Mario World, where Mario must make it from some starting point to the flag in front of a castle, collecting as many coins as possible along the way.

Let $G = (V, E)$ be a directed graph where each vertex $v \in V$ has $C_v \geq 0$ coins. For a walk in $G$, we say that the *number of coins collected by the walk* is the total sum of $C_v$ over all *distinct* vertices $v$ in the walk. (If we visit a vertex $v$ more than once, we still only get $C_v$ coins total.) Given $s, t \in V$, the goal is to compute the maximum number of coins collected by any $(s, t)$-walk.

1. (5 points) Let $G$ be a DAG. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. (5 points) Let $G$ be a general directed graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.24.** Let $G = (L \cup R, E)$ be a bipartite graph. For a set $S \subseteq L$, let $N(S) = \{v \in R : \{u, v\} \in E \text{ for some } u \in S\}$ be the *neighborhood* of $S$. A matching $M$ *covers* $L$ if every vertex in $L$ is an endpoint of some edge in $M$.

1. Prove that if $G$ has a matching covering $L$, then $|N(S)| \geq |S|$ for all $S \subseteq L$.

2. Prove the converse: if $|N(S)| \geq |S|$ for all $S \subseteq L$, then $G$ has a matching saturating $L$.

3. A bipartite graph is *$k$-regular* if every vertex has degree exactly $k$. Prove that every $k$-regular bipartite graph (for $k \geq 1$) has a prefect matching.

**Exercise K.25.** Let $G = (V, E)$ be an undirected graph with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$. The *traveling salesman problem* (TSP) asks for the minimum-length tour in $G$. (A tour starts and ends at the same vertex, and visits each vertex at least once.)

The traveling salesman problem is extremely useful as a prototypical planning and routing problem (broadly construed). It has inspired a number of techniques both in theory and in practice, and also captured the popular imagination. See https://www.math.uwaterloo.ca/tsp/history/index.html for more on the history of this problem.

1. Prove that TSP is NP-Hard (in undirected graphs).[6]

2. Let $\text{OPT}_{\mathsf{TSP}}$ be the minimum TSP value and let $\text{OPT}_{\mathsf{MST}}$ be the weight of the MST with respect to $\ell$. Prove that $\text{OPT}_{\mathsf{MST}} \leq (1 - 1/n)\,\text{OPT}_{\mathsf{TSP}}$.[7]

3. Given TSP is NP-Hard, we might try to get a polynomial time *approximation algorithm* instead. For $\alpha > 1$, an *$\alpha$-approximation* for TSP is a tour of cost at most $\alpha\,\text{OPT}_{\mathsf{TSP}}$. Unlike exact algorithms, a polynomial time $\alpha$-approximation algorithm does not immediately imply an exact polynomial time algorithm for SAT.

   Design and analyze a polynomial time 2-approximation algorithm for TSP.[8]

**Exercise K.26.** The `count-min` data structure allows us to estimate the relative frequency of each element up to an $\epsilon$-additive factor with probability of error $\leq 1/\operatorname{poly}(n)$ with $O(\log(n)/\epsilon)$ space. [9] The original motivation, however, was to also obtain a list of $\epsilon$-heavy hitters. Design and analyze an algorithm that maintains a list of elements, with at any *particular* point in time,[10] with probability of error $\leq 1/n^2$:

1. Contains all of the $\epsilon$-heavy hitters.

2. Only includes $(\epsilon/2)$-heavy hitters.

Your space usage should be comparable to the space used by the `count-min` data structure.[11]

*Additional remark.* The question asks for *one data structure* that satisfies *both the criteria* simultaneously. That is, you should maintain a list $S$ that (a) contains all $\epsilon$-heavy hitters, *and* (b) only includes $(\epsilon/2)$-heavy hitters. The tricky part is that `count-min` only approximates the frequencies. You may want to account for the fact

[6] You can assume that Hamiltonian path in undirected graphs is NP-Hard. (Part 1 of exercise K.119; it was briefly covered at the end of Kent's "Longest Path" lecture..)

[7] *Hint:* You may first want to show that $\text{OPT}_{\mathsf{MST}} \leq \text{OPT}_{\mathsf{TSP}}$.

[8] *Hint:* By part 2, the MST has weight less than $\text{OPT}_{\mathsf{TSP}}$. So two copies of the MST have weight...

[9] Here the elements are integers from $[n] = \{1, \ldots, n\}$, where $n$ is known, and $\epsilon \in (0, 1)$ is an input parameter.

[10] To clarify, what we mean by "particular point in time" is as follows. You have a data structure that is processing data over time. Suppose we suddenly paused the stream and asked you to report your list of heavy hitters. Your algorithm should succeed then and there with probability of error $\leq 1/n^2$. For this criteria, you do not need to know the length of the stream.

[11] You may want to use the `count-min(`$\epsilon$`,`$\delta$`)` data structure as a black box, but you should be clear about your choice of parameters $\epsilon$ and $\delta$.

that an instance of `count-min(`$\epsilon$`,`$\delta$`)` may overestimate the relative frequency of an element by as much as $\epsilon$, which can make a very infrequent element look like an $\epsilon$-heavy hitter.

**Exercise K.27.** Let $G = (V, E)$ be an undirected graph with real-valued edge weights. Recall that the MST minimizes the sum of edge weights over all spanning edge sets of $G$. Consider the problem of computing a spanning edge set $S \subseteq E$ minimizing the maximum edge weight, $\max_{e \in S} w(e)$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

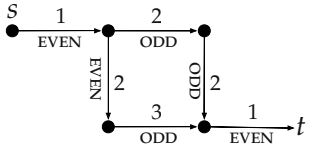
**Exercise K.28. Red light, green light!** The following model is inspired by streetlights which, at a given point in time, allow traffic to enter some streets and block traffic from entering others.

Let $G = (V, E)$ be a directed graph with positive, integer edge lengths $\ell(e)$ for $e \in E$, which represents the amount of time it takes to traverse that edge. (Here time will always be an integer.) Suppose each edge is also annotated as being either "even" or "odd". The model is as follows.

We start at a vertex $s$ at time $x = 0$, and want to get to a vertex $t$ as soon as possible. In general, taking an edge $e$ takes $\ell(e)$ units of time, which is added to $x$. However, we can only take an even edge when the time $x$ is even, and we can only take an odd edge when the time $x$ is odd. We can also wait at a vertex for one unit of time during which $x$ increases by 1.

Loosely speaking, one can imagine each vertex as being like a stop light. An even edge is like a one-way street that traffic can enter only when the time $x$ is even. An odd edge is like a one-way street that traffic can enter only when the time $x$ is odd.

In the example on the right, the $(s, t)$-path along the top takes $1 + 2 + 2 + 1_{(\text{WAIT})} + 1 = 7$ units of time. (Here we annotated the 1's that correspond to waiting). The path along the bottom takes $1 + 1_{(\text{WAIT})} + 2 + 1_{(\text{WAIT})} + 3 + 1 = 9$ units of time.




For $s, t \in V$, consider the problem of computing the minimum amount of time needed to get from $s$ to $t$ in the time model described above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.29.** This exercise is about a simple randomized algorithm for *verifying* matrix multiplication. Suppose we have three $n \times n$ matrices $A, B, C$. We want to

verify if $AB = C$. Of course one could compute the product $AB$ and compare it entrywise to $C$. But multiplying matrices is slow: the straightforward approach takes $O(n^3)$ time and there are (more theoretical) algorithms with running time roughly $O(n^{2.37\cdots})$. We want to test if $AB = C$ in closer to $n^2$ time.

The algorithm we analyze is very simple. Select a point $x \in \{0,1\}^n$ uniformly at random. (That is, each $x_i \in \{0,1\}$ is an independently sampled bit.) Compute $A(Bx)$ and $Cx$, and compare their entries. (Note that it is much faster to compute $A(Bx)$ than $AB$.) If they are unequal, then certainly $AB \neq C$ and we output `false`. Otherwise we output `true`. Note that the algorithm is always correct if $AB = C$, but could be wrong when $AB \neq C$. We will show that if $AB \neq C$, the algorithm is correct with probability at last $1/2$.

1. Let $y \in \mathbb{R}^n$ be a fixed nonzero vector, and let $x \in \{0,1\}^n$ be drawn uniformly at random. Show that $\langle x, y \rangle \stackrel{\text{def}}{=} \sum_{i=1}^n x_i y_i \neq 0$ with probability at least $1/2$. [12]

2. Use the preceding result to show that if $AB \neq C$, then with probability at least $1/2$, $ABx \neq Cx$.[13]

3. Suppose we want to decrease our probability of error to (say) $1/n^2$. Based on the algorithm above, design and analyze a fast randomized algorithm with the following guarantees.

   - If $AB = C$, then it always reports that $AB = C$.
   - If $AB \neq C$, then with probability of error at most $1/n^2$, it reports that $AB \neq C$.

   (Your analysis should include the running time as well.)[14]

**Exercise K.30.** Let $G = (V, E)$ be a connected undirected graph with positive edge weights $w : E \to \mathbb{R}_{>0}$, and let $U \subseteq V$ be a set of terminals. Recall that the minimum weight Steiner tree problem asks for a minimum weight tree in $G$ connecting all terminals in $U$.

Recall that the (shortest path) distances between vertices define a metric over $V$. The *metric closure* of $G$ over $U$ is the complete graph on $U$ with edge weights given by the distances in $G$.

[12] *Hint: Suppose for simplicity that the last coordinate of $y$ is nonzero. It might help to imagine sampling the first $n-1$ bits and computing the partial sum $S_{n-1} = \sum_{i=1}^{n-1} x_i y_i$ first, before sampling $x_n$ and adding $x_n y_n$. Formally your analysis may involve some conditional probabilities. (And what about the case where $y_n = 0$?)*

[13] Even if you haven't solved part 1 you may assume it to be true.

[14] *Hint:* How can you decrease the probability of error to 1/4? 1/8? 1/16?

1. Design and analyze an algorithm to compute the metric closure of $G$ over $U$.
2. Let $\text{OPT}_{\mathsf{ST}}$ denote the minimum weight of a Steiner tree connecting $U$ in $G$, and let $\text{OPT}_{\mathsf{MST}}$ denote the weight of the MST of the metric closure of $G$ over $U$. Prove that $\text{OPT}_{\mathsf{MST}} \leq 2\,\text{OPT}_{\mathsf{ST}}$.
3. Building on parts 1 and 2, design and analyze a polynomial time 2-approximation algorithm for the minimum weight Steiner tree problem in undirected graphs.

**Exercise K.31.** Section 22.5.1 showed how to use two (immutable) stacks to implement a queue with $O(1)$ amortized performance. Here we will develop a more sophisticated, *double-ended queue* using three immutable stacks. (It's mostly using two stacks, but occasionally needs a third stack as a buffer.)

A double ended queue is like a queue except we can insert and delete from both ends of the queue. Here we designate one end as the "front" and the other end as the "back", and define the following four operations.

1. `push-front(`$x$`)`: inserts element $x$ at the front of the queue.
2. `push-back(`$x$`)`: inserts element $x$ at the back of the queue.
3. `pop-front()`: removes the first element of the queue.
4. `pop-back()`: removes the last element of the queue.

(The data structure throws an error if the user calls either `pop` operation and there are no elements remaining.)

1. The natural extension to the two-stack approach is to flip all the elements over whenever you are popping from a side with an empty stack. Prove that this approach does *not* get $O(1)$ amortized time.
2. Design and analyze a different data structure, using at most 3 immutable stacks, that obtains $O(1)$ amortized time for each operation. (Our solution is somewhat similar to the two-stack approach, but the modification requires using a third stack on occasion as a buffer.)

**Exercise K.32.** Consider the following generic recursion.

$$T(n) = \alpha T(\beta n) + n^{\gamma}$$
$$T(1) = \delta$$

where $\alpha, \beta, \gamma, \delta > 0$ are fixed constants. Prove each of the following using the recursion tree method.

1. For any $\beta < 1$, $T(n)$ is at most a polynomial in $n$. (Even if, say, $\alpha = 2^{260}$. Here $\alpha, \beta, \gamma, \delta$) may appear in the exponent of the polynomial.) For simplicity, you may assume that $\alpha$ is an integer.

2. Show, using the recursion tree method, that if $\alpha\beta^\gamma < 1$, then $T(n) \leq O(n^\gamma)$.

3. Show, using the recursion tree method, that if $\alpha\beta^\gamma = 1$, and $\beta < 1$, $T(n) \leq O(n^\gamma \log(n))$.

**Exercise K.33. Who needs friends to play Set?** The card game *Set* has 81 different cards varying in four features across three possibilities for each kind of feature. The four features are:

1. The shape, which is either a diamond, a squiggle, or an oval.
2. The number of shapes, which is either 1, 2, or 3.
3. The color of the shape, which is either red, green, purple.
4. The shading of the shape, which is either solid, striped, or open.

There are multiple variations of Set but they all are based on trying to form particular "sets" of three cards. A set consists of three cards satisfying *all* of these conditions:

1. They all have the same number or have three different numbers.
2. They all have the same shape or have three different shapes.
3. They all have the same shading or have three different shadings.
4. They all have the same color or have three different colors.

Loosely speaking, the rules of Set are summarized by: If you can sort a group of three cards into "two of _____ and one of _____", then it is not a set.

For example, the three cards on the right form a set. We have one card of each color, one of each shape, one of each shading, and one of each number. (In case this is printed in black and white, we mention that the first card is green, the second is purple, and the third is red.)

We will study a solitaire version of Set. Solitaire Set starts with 3 decks of cards. Each round you choose one of the following options.

1. If the three cards on top of the deck form a set, you may remove all three from the top of their decks, and set them aside.

2. You may discard one card from the top of one of the decks, revealing a new card underneath (unless that deck is empty).

(Even if the top three cards form a set, you can choose to discard one of them instead.) The goal is to form as many sets as possible by the time you empty all the decks.

We will consider the game as an optimization problem where we also know the full sequence of cards in the deck. The three decks are represented by three arrays $A[1..m]$, $B[1..n]$, and $C[1..p]$.[15] The cards are listed from top to bottom in each deck. *For simplicity, we assume access to a constant-time subroutine* `IsItASet(i,j,k)` *that takes as input three indices $i, j, k$ and outputs* `True` *if the cards $(A[i], B[j], C[k])$ form a set, and* `False` *if they do not.*

The problem is to compute the maximum number of sets that can be obtained in a game of Solitaire Set over the decks $A[1..m]$, $B[1..n]$, and $C[1..p]$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.34.** Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two directed graphs with $n_i = |V_i|$ vertices and $m_i = |E_i|$ edges each. Suppose we have a labeling $\ell : V_1 \cup V_2 \to \mathcal{A}$, where $\mathcal{A}$ is an alphabet. (Different vertices may share the same letter.) A *common path* is defined as a path $p_1$ in $G_1$ and a path $p_2$ in $G_2$ such that the letters along the paths are the exact same. (The first vertices of $p_1$ and $p_2$ have the same letter, the second vertices have the same letter, etc. $p_1$ and $p_2$ must have the same length.) Consider the problem of computing the length of a longest common path in $G_1$ and $G_2$.

1. (5 points) Suppose $G_1$ and $G_2$ are both DAGs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. (5 points) Now suppose $G_1$ and $G_2$ are general directed graphs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.35.** Let $G = (V, E)$ be a directed graph. We say two vertices $s, t \in V$ are *half-connected* if either $s$ can reach $t$ or $t$ can reach $s$. We say that the graph $G$ is *half-connected* if every pair of vertices is half-connected. We say that two vertices $s$ and $t$ are *strictly half-connected* if either $s$ can reach $t$, or $t$ can reach $s$, but not both. We say that $G$ is strictly half-connected if every pair of vertices is strictly half-connected.

1. Consider the problem of deciding if $G$ is half-connected. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

[15] The cards in these decks are not distinct; a particular card may have multiple copies. So, for example, there may be many more than 81 cards total.

2. Consider the problem of deciding if $G$ is strictly half-connected. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.36.** Each of the following problems takes as input an array $A[1..n]$ of integers and asks for an optimal subsequence.[16] Design and analyze an algorithm for each, addressing Items 1–5 from Section 5.2.[17]

1. A sequence of numbers $x_1, \ldots, x_k$ is (strictly) increasing if $x_i < x_{i+1}$ for $i = 1, \ldots, k-1$. Compute the length of the longest increasing subsequence of $A$.

2. A sequence of numbers $x_1, \ldots, x_k$ is *convex* if $x_{i+1} - x_i \geq x_i - x_{i-1}$ for $i = 2, \ldots, k-1$. Compute the length of the longest convex subsequence of $A$.

3. Compute the maximum sum of any increasing subsequence of $A$.

4. Compute both:

   - the length of the longest increasing subsequence of $A$ where the sum of integers is even,
   - the length of the longest increasing subsequence where the sum of integers is odd.

5. Recall that the Fibonacci sequence

   $$1, 1, 2, 3, 5, 8, \ldots$$

   can be characterized as a sequence starting with 1, 1 and such that every successive number is the sum of the previous two numbers. ($1+1=2$, $1+2=3$, $2+3=5$, ...) Generalizing this notion, we say that a sequence $y_1, y_2, \ldots, y_k$ is *sub-Fibonacci* if $y_i \leq y_{i-2} + y_{i-1}$ for $i = 3, 4, \ldots, k$. That is, each successive number is at most the sum of the previous two.

   Compute the length of the longest increasing and sub-Fibonacci subsequence of $A$.

[16] A "subsequence" of $x_1, \ldots, x_n$ is a sequence of the form $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$.

[17] In particular, no proof is required. We will interpret your recursive spec as the inductive hypothesis of a proof by induction, and fill in the rest of the details ourselves. That said, some additional commentary may help the grader, especially in the event of partial credit.

6. Suppose each entry in $A$ is also colored red, white, or blue. We say that a sequence is *American* if the colors alternate red, white, blue, red, white, blue, ... The first number in the sequence can be any color. Compute the length of the longest increasing American subsequence of $A$. (You can assume that you can look up the color of $A[i]$, for any $i \in [n]$, in constant time.)

7. A sequence $x_1, \ldots, x_k$ is a *palindrome* if the reversed sequence is the same; i.e., $(x_1, \ldots, x_k) = (x_k, \ldots, x_1)$. For example,

   *mom*, *dad*, *racecar*, and *gohangasalamiimalasagnahog*

   are all palindromes. Compute the length of the longest palindrome subsequence of $A$.

8. Compute the length of the shortest supersequence of $A$ that is a palindrome.[18]

9. Suppose each entry of $A$ is colored red, white or blue. We say that a sequence is *un-American* if the colors in the sequence never cycle through the American dream; that is, no three consecutive entries have colors that form any of the following three sequences: (red, white, blue), (white, blue, red), or (blue, red, white). Compute the length of the longest increasing un-American subsequence of $A$. (You can assume that you can look up the color of $A[i]$, for any $i \in [n]$, in constant time.)

10. (Just for fun) Invent your own fun and interesting version of a dynamic programming problem over $A$.

**Exercise K.37.** Let $A[1..m]$ and $B[1..n]$ be two sorted arrays of comparable elements. Given an integer $k \in \{1, \ldots, m+n\}$, the goal is to find the $k$th largest element among the combined elements of $A$ and $B$. For simplicity you make assume that all the elements are distinct.[19]

1. Define a recursive spec for a recursive algorithm to solve this problem.

2. Show that if either $m$ or $n$ is $\leq 5$ then we can find the $k$th element in $O(1)$-time by direct means. (This is to help declutter your implementation and avoid fencepost errors.)

[18] $x_1, \ldots, x_k$ is a *supersequence* of $y_1, \ldots, y_\ell$ if $y_1, \ldots, y_\ell$ is a subsequence of $x_1, \ldots, x_k$.

[19] By assuming standard data structures keeping track of the offset and length, you can get (a pointer to) the subarray $A[i..j]$ of an array $A[1..m]$ in $O(1)$ time.

3. Give a recursive algorithm implementing your recursive spec. You may simply refer to the previous part for your base case.

4. Prove your algorithm is correct by induction.

5. Analyze the running time of your algorithm.

**Exercise K.38.** Recall that the Economist recently reported[20] that the number of toll roads in America is increasing rapidly. It used to be hard enough to find the shortest path. Now you have to do it within a budget.

Right now, drivers pay to access 6,300 miles of Americas 160,000 miles (or so) of highway, but that number is going up. Here in Indiana, in June, the state passed a law giving the state authority to raise tolls on all its existing interstate highways. The Economist says that this trend is broadly due to a parallel trend of decreasing taxes on gasoline, which used to pay a larger share of the cost for maintaining roads and highways.

Let $G = (V, E)$ be a directed graph with nonnegative integer edge lengths $\ell : E \to \mathbb{Z}_{\geq 0}$ and *nonnegative integer edge costs* $\texttt{cost} : E \to \mathbb{Z}_{\geq 0}$, which model toll roads. We define the *cost* of a walk in $G$ as the sum of edge costs (with repetition) along that walk. Let $k \in \mathbb{N}$ be a given budget. Consider the problem of computing the length of the shortest $(s, t)$-walk that costs at most $k$. (Unlike the midterm, we do not assume that $k$ is polynomially bounded by $m$ and $n$.) For this problem, either design and analyze a polynomial time algorithm, or prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise K.39.** Let $G = (V, E)$ be an undirected graph with edge weights $w : E \to \mathbb{R}$. Recall that the minimum weight spanning tree can be computed in polynomial time by a number of different algorithms. Suppose we imposed an additional constraint that the spanning tree has maximum (unweighted) degree at most $\Delta$ for some given integer $\Delta \in \mathbb{N}$.

1. (5 points) Consider the problem of computing the minimum weight spanning tree with maximum degree at most 2. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

[20] *Toll roads are spreading in America*, December 18, 2025. https://www.economist.com/united-states/2025/12/18/toll-roads-are-spreading-in-america

2. (5 points) Consider the problem of computing the minimum weight spanning tree with maximum degree at most $\Delta$, for $\Delta \geq 3$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.40.** Above we observed that two degree $n$ polynomials $p(x)$ and $q(x)$ can be multiplied in $O(n \log n)$ time. Consider the case when the degrees are very unbalanced; say, $p(x)$ has degree $n$ and $q(x)$ has degree $k$ for $k \ll n$. Design and analyze an algorithm computing the product $p(x)q(x)$ in $O(n \log k)$ time.

**Exercise K.41.** Let $A[1..n]$ be an array of integers in the set $\{1..n\}$, sorted in increasing order. (There may be repeats.) A *fixed point* is an index $i \in [n]$ such that $A[i] = i$.

1. (2 pt.) Prove the following by induction on $j - i$: *for all pairs of indices $i, j$ such that $1 \leq i \leq j \leq n$, $i \leq A[i]$, and $A[j] \leq j$, there is a fixed point $k$ in the range $i \leq k \leq j$.*

   This claim shows in particular that $A$ has a fixed point.

2. (2 pt.) Moving onto developing algorithms, give a recursive spec for the problem of finding a fixed point in $A$.[21]

3. (2 pt.) Give a recursive algorithm (the faster the better) implementing your recursive spec.

4. (2 pt.) Prove your algorithm is correct by induction. (There may or may not be some redundancy in your argument with part 1. You may also use the mathematical fact established in part 1.)

5. (2 pt.) Analyze the running time of your algorithm.

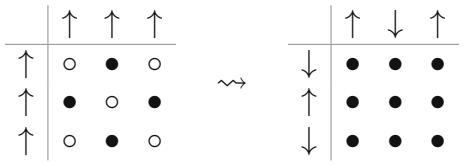
**Exercise K.42. Let there be light.** $n^2$ lightbulbs are arranged in an $n \times n$ grid. Each row has a switch and each column has a switch, for $2n$ switches in total. Each switch can be `up` or `down`. Flipping a row switch toggles every bulb in that row. Flipping a column switch toggles every bulb in that column.

The input is an $n \times n$ array $A$ describing the state of each lightbulb (`on` or `off`) when every switch is `up`. (In general, some lightbulbs may be on, while others are off,

[21] There may be more than one fixed point; any fixed point is fine.

initially.) The goal is to decide if there is a configuration of switches that turns on all the lightbulbs simultaneously.

For example, consider the $3 \times 3$ instance below ($\bullet$ = `on`, $\circ$ = `off`), with row switches on the left and column switches on top. The configuration on the right turns on all the lightbulbs.



For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.43.** Recall that given a set $P$ of $n$ points in $\mathbb{R}^2$, we can compute the minimum distance between any pair of points in $P$ in $O(n \log n)$ time. Consider now the problem of computing the second smallest distance between any pair of points in $P$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.44.** Problem 7 of exercise K.36 considered the problem of computing the longest palindrome subsequence. Here we recall that a palindrome is a string spelled the same forwards and backwards, such as "racecar". Here we consider the problem of computing the maximum weight palindrome subsequence where there characters of the input string are each given weights.

For example, consider the array of characters below where we annotate each character with its weight in the subscript.

$$\mathtt{x}_8\ \mathtt{r}_5\ \mathtt{y}_1\ \mathtt{a}_3\ \mathtt{c}_7\ \mathtt{z}_4\ \mathtt{e}_2\ \mathtt{c}_4\ \mathtt{x}_{-5}\ \mathtt{a}_5\ \mathtt{r}_{-1}$$

Above, `racecar` is a palindrome subsequence, with total weight $22 = 5 + 3 + 7 + 2 + 4 + 5 - 1$.

The input consists of a string $A[1..n]$ and an array $W[1..n]$ of real-valued weights. Each $W[i]$ gives the weight of character $A[i]$. The goal is to compute the maximum total weight of any palindrome subsequence of $A[1..n]$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.45.** The following problems consider variations for the longest path problem that instead ask for longest walks (with varying definitions of "longest"). Here we recall that a walk may repeat vertices and edges, whereas a path cannot. Let $G = (V, E)$ be a directed graph with $m$ edges and $n$ vertices. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

1. Compute the length of the longest walk in $G$.
2. Compute the maximum number of distinct vertices of any walk in $G$.
3. Compute the maximum number of distinct edges of any walk in $G$.

**Exercise K.46.** A polygonal path is a sequence of line segments joined end-to-end; the endpoints of these line segments are called the vertices of the path. The length of a polygonal path is defined as the sum of the lengths of its segments. A polygonal path with vertices $(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)$ is *monotonically increasing* if $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for every index $i$ – informally, each vertex of the path is above and to the right of its predecessor.

Consider the problem of computing the length of the longest monotonically increasing path among a set $S$ of $n$ points. Here the input consists of a set $S$ of $n$ points in the plane, represented as two arrays $X[1..n]$ and $Y[1..n]$ (for the $x$-coordinates and $y$-coordinates, respectively). You may assume access to a subroutine `Length(`$x$`,`$y$`,`$x'$`,`$y'$`)` that returns the length of the segment from $(x, y)$ to $(x', y')$ in $O(1)$ time. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.47.** For each of the recursive specifications below, design a recursive algorithm implementing the specification. (No proof or analysis is needed.)[22]

1. `subset-sum(`$x_1, \ldots, x_n$`,` $T$`)`: Given $n$ integers $x_1, \ldots, x_n \in \mathbb{Z}$, and an additional integer $T$, returns `true` if there is a subset of indices $S \subseteq [n]$ such that $\sum_{i \in S} x_i = T$, and `false` otherwise.

[22]Your algorithms will not be very fast, but their correctness should follow from an induction argument using the recursive spec as the induction hypothesis.

You are welcome to describe your algorithm using high level steps as long as they are clearly easy to implement and would take linear time; e.g., 'let $G_1$ be the graph obtained by removing such and such vertices and such and such edges." See the sample solutions on page 60.

Some of these problems have well-known efficient algorithms. Efficiency is besides the point; the goal here is to rapidly get some practice thinking recursively.

Lastly, do not change the recursive spec.

2. `min-vertex-cover(`$G = (V, E)$`)`: Given an undirected graph $G = (V, E)$, return the minimum size of any vertex cover of $G$.

   (A vertex cover is a set of vertices $S \subseteq V$ that includes at least one endpoint of every edge $e \in E$.)

3. `max-matching(`$G = (V, E)$`)`: Given an undirected graph $G = (V, E)$, return the maximum size of any matching.

   (A matching is a set of edges $M \subseteq E$ with disjoint endpoints.)

4. `max-independent-set(`$G = (V, E)$: Given an undirected graph $G = (V, E)$, return the maximum size of any independent set of vertices.

   (Given an undirected graph $G = (V, E)$, an *independent set* is a set of vertices $S \subseteq V$ such that no two vertices $u, v \in S$ are connected by an edge.)

5. `longest-increasing-subsequence-including-first(`$x_1, \ldots, x_n$`)`: Given a sequence of numbers $x_1, \ldots, x_n$, return the length of the longest (strictly) increasing subsequence of $x_1, \ldots, x_n$ over all subsequences that include $x_1$. (You may assume $n \geq 1$.)

   (A subsequence of $x_1, \ldots, x_n$ is a sequence of the form $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$, where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. A sequence of numbers $y_1, \ldots, y_\ell$ is strictly increasing if $y_i < y_{i+1}$ for $i = 1, \ldots, \ell - 1$.)

6. `reachable(`$G = (V, E)$`,`$s$`,`$t$`)`: Given a directed graph $G$ and two vertices $s, t \in V$, return `true` if $s$ can reach $t$ in $G$, and `false` otherwise.

   ($s$ can reach $t$ if either $s = t$ or there is a sequence of (directed) edges of the form $(s, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, t)$. Note that the endpoint of one edge is the initial point of the next edge. This is also called a *(directed) walk* in $G$ from $s$ to $t$.)

7. `partition(`$x_1, \ldots, x_n$`)`: Given $n$ integers $x_1, \ldots, x_n \in \mathbb{Z}$, returns `true` if there is a subset of indices $S \subseteq [n]$ such that $\sum_{i \in S} x_i = \sum_{i \in [n] \setminus S} x_i$, and false otherwise.[23]

**Exercise K.48.** Recall the dominating set problem from Section 13.3. Here we will consider the weighted version where the vertices are given positive weights, and the goal is to compute the minimum weight dominating set. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

[23] This might be the trickiest one. No, you cannot reduce to subset sum. You need to implement *this* recursive spec, recursively.

1. The minimum weight dominating set problem for intervals, *with the additional assumption that* no two intervals are nested.

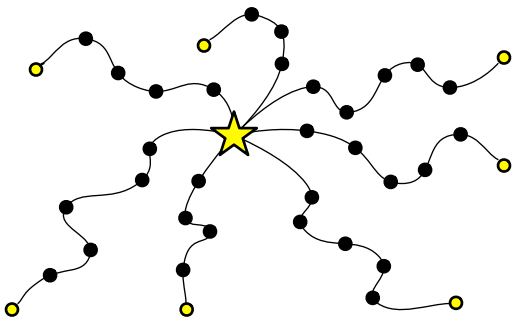
   To state it more precisely: the input consists of $n$ weighted intervals $\mathcal{I}$. The non-nested assumptions means that for any two intervals $I, J \in \mathcal{I}$, we never have $I$ contained in $J$ or $J$ contained in $I$.

   The goal is to compute the minimum weight subset $S \subseteq \mathcal{I}$ of intervals such that every interval in $\mathcal{I}$ is either in $S$ or overlaps some interval in $S$.

   - (For 1 pt. extra credit) Extend your algorithm to general intervals.[24]

2. The minimum weight dominating set problem in trees.

**Exercise K.49.** A *star* is defined as a tree where at most one vertex has degree $> 2$. This special vertex is called the *center* of the star. (A star can have no particular center when it is just a path.)



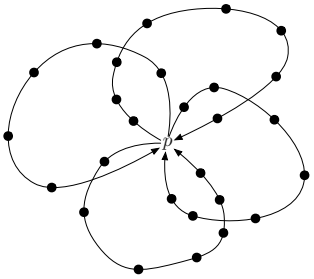
Let $G = (V, E)$ be an undirected graph and $L \subseteq V$ a fixed set of vertices. Here we are interested in stars (as subgraphs of $G$, and centered anywhere) with as many leaves in $L$ as possible. Consider the problem of computing the maximum number of leaves in $L$ that can be obtained by any star in $G$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.50.** According to Wikipedia, a pinwheel is a simple child's toy made of a wheel of paper or plastic curls attached at its axle to a stick by a pin. It is designed to spin when blown upon by a person or by the wind.

[24] Of course, anyone who has already solved the general case automatically solves the special case where no two intervals are nested.

A similar toy had developed independently in Polynesia (known as pekapeka or peʻapeʻa) using either coconut palm leaflets or strips of pandanus leaves;[25][26] in colder climates like that of New Zealand (the toy also called pepepe in Māori), phormium leaves are used.[27]

Let $G = (V, E)$ be a directed graph. A *pinwheel* (in $G$) is defined as a collection of directed cycles all containing a common vertex $p$ and vertex disjoint otherwise. $p$ is called the *pin* of the pinwheel, and each cycle is called a *curl*. Consider the problem of computing the maximum number of curls in any pinwheel. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.



**Exercise K.51.** Consider the following algorithm that claims to compute the MST (in a connected graph).

`decremental-MST`

1. Sort and index $E = \{e_1, \ldots, e_n\}$ in decreasing order of weight.
2. For $i = 1, 2, \ldots, n$:
   A. Unless $E - e_i$ is disconnected:
      1. $E \leftarrow E - e_i$.
3. Return $E$.

Either

1. Describe an input graph where `decremental-MST` fails to find the MST, or
2. Prove that `decremental-MST` always returns the minimum weight spanning tree.

For simplicity you may focus on the setting where the edge weights are distinct.

**Exercise K.52.** Our department has a large catalog of classes and a handful of course requirements you have to satisfy to graduate. The requirements seem to be particularly complicated for undergraduates. Each requirement is defined by a subset

[25]Koch, Gerd (1984). The Material Culture of Tuvalu. Suva: Institute of Pacific Studies, University of the South Pacific. p. 162. ISBN 9820202051.

[26]Te Rangi Hīroa (1930). Samoan Material Culture. Bernice P. Bishop Museum. p. 552.

[27]Beattie, Herries (1994). Traditional Lifeways of the Southern Maori: The Otago University Museum Ethnological Project, 1920. University of Otago Press. p. 68. ISBN 978-0-908569-79-3.

of classes and a number specifying the number of courses you have to take from this subset.

Formally, let the courses be indexed 1 through $n$. We have $L$ requirements $(S_1, k_1), \ldots, (S_L, k_L)$, each consisting of a set $S_i \subseteq [n]$ and an integer $k_i \in \mathbb{N}$. For each $i$, you have to take at least $k_i$ classes from $S_i$. Let $T \subseteq [n]$ be the set of classes you've already taken. Let $m = \sum_i |S_i|$ denote the total size of all the sets.

Now we have two possible interpretations of the rules. In the first version, a single class can only count towards a single requirement. In the second version, a single class can count towards any number of requirements. For example, suppose you are required to take $(k_1 = 2)$ classes from $S_1 = \{A, B, C\}$ and $(k_2 = 2)$ classes from $S_2 = \{C, D, E\}$, and you have already taken $T = \{B, C, D\}$. In the first version, you do not have enough classes to graduate; in the second version, you do have enough classes to graduate.

Here we have related but slightly different problems for the two systems.

1. (5 points) Suppose a single class can only count towards a single requirement. Consider the problem of deciding if you have already taken enough classes to graduate. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. (5 points) Now suppose a single class can count towards any number of requirements. Consider the problem of identifying the minimum number of additional classes that need to be taken to satisfy all the requirements. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.53.** Last week you decided to clean up your toolshed, but you compulsively made the following silly mistake. Originally you had $n$ matching pairs of nuts and bolts of different sizes, but you decided to organize the nuts and bolts separately into a box full of nuts and a box full of bolts, splitting up all the pairs of nuts and bolts in the process. So now you have $n$ nuts, and $n$ bolts, such that each nut fits a distinct bolt and vice-versa, but you don't know which nut goes with which bolt.

The bolts all look pretty similar, so you can't compare two bolts directly with each other and tell which one is bigger. Likewise you cannot compare the nuts to each other. However, you can try to fit one nut to one bolt, and see if either:

1. The nut fits the bolt.
2. The nut is too big for the bolt.
3. The nut is too small for the bolt.

We will treat one of these nut-to-bolt tests as a single operation. Your goal is to match up all nuts and bolts. Of course you can compare every pair of nut and bolt in $O(n^2)$ time, but can you do better?

Design and analyze an algorithm, as fast as possible, to recover all matching pairs of nuts and bolts.

**Exercise K.54.** Let $A[1..m]$ be an array of integers, and let $T$ be an additional number. We say that a sequence of numbers

$$y_1, y_2, \ldots, y_k$$

is *weakly increasing* if $y_1 \leq y_2 \leq \cdots \leq y_k$. The problem is to find the length of the longest increasing subsequence of $A$ where the numbers sum to $T$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.55.** Use recursion trees to solve the following recurrences.

1. $T(n) = 3T(n/3) + 6n$ and $T(1) = 2$.
2. $T(n) = 2T(n/3) + 4n$ and $T(1) = 7$.
3. $T(n) = 4T(n/3) + n$ and $T(1) = 11$.

**Exercise K.56.** Let $G = (V, E)$ be an undirected graph, with edge weights $w : E \to \mathbb{R}_{>0}$. ($G$ can have parallel edges.) A *cut* is a set of edges of the form

$$\delta(S) \stackrel{\text{def}}{=} \{\{u, v\} \in E : u \in S,\, v \notin S\}$$

for a set of vertices $S \subseteq V$. Note that $\delta(S) = \delta(V \setminus S)$. $S$ and $V \setminus S$ are called *sides* of the cut. The *weight* of a cut $\delta(S)$ is the sum of weights over all edges $e \in \delta(S)$.

1. The weighted max cut problem is to compute the cut $\delta(S)$ of maximum total weight. The goal of this problem is to show that it is as hard as SAT to find a polynomial time algorithm, via a reduction from max 2-sat. The reduction is somewhat involved, so we will describe most of it and then ask you to fill in the remaining details and complete the proof.

   Suppose we have an instance of max 2-SAT defined by a boolean formula $f(x_1, \ldots, x_n)$ in CNF with exactly two variables per clause. Let $C_1, \ldots, C_m$ be the $m$ clauses of $f$.

   We create a graph $G = (V, E)$ as follows. For the vertices $V$:

   - For each variable $x_i$, we have vertices representing the literals $x_i$ and $\bar{x}_i$.
   - We also have two vertices representing `true` and `false`.

   For the edges $E$:

   - For each variable $x_i$, we add an edge between $x_i$ and $\bar{x}_i$ with weight $W$, for some value $W > 0$ TBD.
   - We also have an edge between `true` and `false` of weight $W$.
   - For each clause $C_i$ of the form $(a_i \vee b_i)$, where $a_i$ and $b_i$ are literals, we add the edges $\{a_i, b_i\}, \{a_i, \texttt{false}\}, \{b_i, \texttt{false}\}$, each with weight 1.

   The high-level idea is that, for a sufficiently large value of $W$, any maximum weight cut $\delta(S)$ should contain all edges of weight $W$. In particular, (the vertices for) `true` and `false` will be on opposite sides of the cut, and the (vertices corresponding to the) literals $x_i$ and $\bar{x}_i$ will be on opposite sides of the cut for each variable $x_i$. When a literal is on the same side as `true`, we interpret that as assigning the literal to be true; when a literal is on the same side of the cut as `false`, we interpret that as assigning the literal to be false.

   Given the construction described above, it remains to choose $W$ appropriately and prove that this gives a reduction from max 2-SAT to max-cut.

2. Now consider the unweighted version of the problem, where all edges have weight 1. Here $G$ is allowed to have parallel edges. Either (a) design and analyze a polynomial time algorithm for unweighted max-cut, or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT. For option (b), if you are building on / adjusting the construction from problem 1, you can explain the differences from that construction and argument, instead of rewriting the entire thing.

3. Next we consider the problem of deciding if $E = \delta(S)$ for some set $S \subseteq V$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

4. Finally, consider the problem of deciding if there is a cut of weight at least half of the total weight of all the edges. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.57.** Consider an instance $(G = (V, E), s, t, c)$ of $(s, t)$-maximum flow with integral capacities, and suppose you have already computed a maximum integral flow $f$.

1. Suppose we increase the capacity of an edge by 1. Design and analyze an algorithm, as fast as possible, to compute the maximum flow in the updated graph.

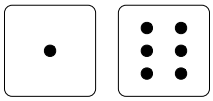
2. Suppose we decrease the capacity of an edge by 1. Design and analyze an algorithm, as fast as possible, to compute the maximum flow in the updated graph.

**Exercise K.58.** A *Hamiltonian cycle* is a cycle that visits every vertex exactly once.

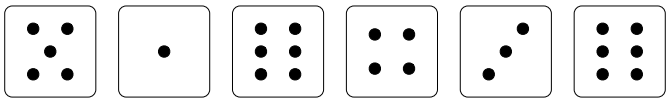
1. Consider the problem of deciding whether a given directed graph has a Hamiltonian cycle. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. Give a polynomial time reduction from the Hamiltonian cycle problem to the Hamiltonian path problem (showing directly that a polynomial time algorithm for Hamiltonian path implies a polynomial algorithm for Hamiltonian cycle).

**Exercise K.59.** Recall that when we roll six-sided dice, the dice samples an integer between 1 and 6 uniformly at random. Let us call an unordered pair of dice "lucky" if one of them is a 1 and the other is a 6.



If we roll 6 independent six-sided dice, how many lucky pairs do we expect? Note that a single dice may appear in more than one lucky pair. For example, the following roll of six dice has 2 lucky pairs amongst them.



**Exercise K.60.** Here we consider an extension of shortest $(s,t)$-walks where one has to visit a family of vertices specified by the input. The input consists of a directed graph $G = (V, E)$ with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$, as well as a list of vertices $x_1, \ldots, x_k \in V$. The high-level goal in both of the following problems is to compute the length of the shortest walk that visits all $k$ vertices.

1. Suppose you are allowed to visit $x_1, \ldots, x_k$ in any order. Consider the problem of computing the length of the shortest walk visiting all of $x_1, \ldots, x_k$ in any order. (You may assume no vertices repeat in $x_1, \ldots, x_k$.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. Suppose you have to visit $x_1, \ldots, x_k$ in the listed order. (Here vertices may repeat, but for simplicity you may assume that any two consecutive vertices $v_i$ and $v_{i+1}$ are distinct.) Consider the problem of computing the length of the shortest walk visiting $x_1, \ldots, x_k$ in order. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.61.** Consider the edit distance problem from Section 5.1. In Section 5.1, insertions, deletions, and substitutions were all treated equal. Suppose instead that we had different costs $\alpha, \beta, \gamma > 0$ for each insertion, deletion, and substitution, respectively. Show how to modify the algorithm from Section 5.1 for these nonuniform costs. This model is useful in biological settings where $\alpha, \beta, \gamma$ reflect prior assumptions on the likelihood of each operation.

**Exercise K.62.** Given a graph $G = (V, E)$, a *clique* is a set of vertices $S \subseteq V$ such that every pair of vertices in $S$ is connected by an edge. (This is sort of like the opposite of an independent set.) When the vertices are weighted by $w : V \to \mathbb{R}_{>0}$, one can ask for the maximum weight clique in the graph.[28]

[28]Note: we updated the problems so that one only needs to return the maximum weight of any clique, as opposed the actual clique itself, for the sake of simplicity.

1. Consider the problem of finding the largest weight of any clique from a set of weighted intervals. (Here a clique of intervals is a set of intervals where every two intervals are overlapping.) The input consists of $n$ intervals $I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]$, and you can assume that you can look up the weight of an interval in constant time. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. Consider the problem of finding the largest weight of any clique in a tree. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

3. Consider the problem of finding the largest weight of any clique in a graph $G$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.[29]

**Exercise K.63.** Let $A$ and $B$ be two events with $\mathbf{P}[A] + \mathbf{P}[B] = 1$. Prove or disprove: $\mathbf{P}[A \vee B] = 1$ iff $\mathbf{P}[A \wedge B] = 0$.

**Exercise K.64.** Recall that a (binary) min-heap maintains a collection of elements and supports insertion and removing the minimum element in $O(\log n)$ worst-case time. Here we will develop and analyze a more sophisticated data structure called a *binomial heap*, which obtains $O(1)$ amortized time for insertion and $O(\log n)$ amortized time for deletion.

At a high-level, a binomial heap maintains a family of rooted trees over its elements, where each tree is in min-heap order. (Each node is less than or equal to any of its children). We will call each of these individual trees "heaps". The heaps in a binomial heap have a specific structure based on the following notion of "ranks".

- A rank 0 heap consists of a single node.
- For $i > 0$, a rank-$i$ heap consists of two rank $i - 1$ heaps, where one heap is the child of the root of the other.

Two rank-$i$-heaps can be merged into a rank-$(i + 1)$-heap in constant time. Binomial heaps maintain the invariant that there is at most one heap of any rank $i$.

Now we give a high-level sketch of the `insert` and `remove-min` subroutines. Both have a postprocessing step that fixes the heap invariant. You will flesh out the implementation details in part 4 below.

[29] Hint: Consider the "complement graph" $G' = (V, E')$ where there is an edge $\{u, v\}$ iff $\{u, v\}$ *is not* an edge in $G$.

`insert(`$x$`)`

*/* High-level sketch; cf. part 4. */*

1. Create a rank 0 heap with node $x$.
2. Restore the heap invariant by repeatedly merging heaps of the same rank.

`remove-min`

*/* High-level sketch; cf. part 4. */*

1. Scan the roots of all the heaps. Let $H$ be the heap with the smallest root, let $x$ be the root.
2. Remove $H$ from the collection of heaps, and add the sub-heaps corresponding to the children of $x$.
3. Fix the heap invariant by repeatedly merging heaps of the same rank.
4. Return $x$.

Implement and analyze the binomial heap data structure via the following steps.

1. How big is a heap of rank $i$?
2. Suppose there are $n$ elements. How many heaps are in the binomial heap?
3. Prove that for $i > 0$, a heap of rank $i$ consists of a root node with $i$ children, where the $j$th child is a heap of rank $j - 1$.
4. Flesh out the implementation details for `insert` and `remove-min`[30]. Prove that, for your implementation, `insert(`$x$`)` has $O(1)$ amortized time and `remove-min` has $O(\log n)$ amortized time.[31]

**Exercise K.65.** Let $P$ be a set of $n$ points in $\mathbb{R}^2$.[32] Suppose you want to compute a list of all pairs of points with distance with a factor of 2 of the minimum distance between all pairs of points. Design and analyze an algorithm for this problem.

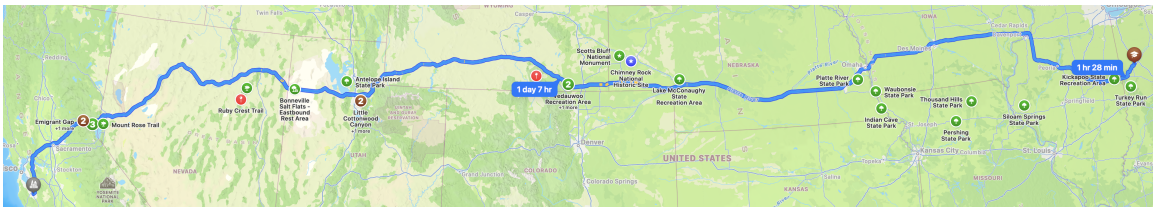
(Food for thought: what does your running time say about the total number of pairs of points with distance within a constant factor of the minimum distance between all pairs of points?)

[30] How do we store the heaps? How can we efficiently "merge" the heaps of the same rank? Etc.

[31] Note that one could also show that binomial heaps have $O(\log n)$-worst case time for both operations.

[32] For simplicity you may assume the points are in general position; that is, no two points have the same $x$-coordinate or the same $y$-coordinate.

**Exercise K.66.** Imagine you are planning a long road trip to San Jose, CA.




This time you haven't fixed your route and you're not so obsessed with gas stations. You decided you want to do some sightseeing along the way and you've marked out some points of interest across America. Your goal is to visit as many of these points as possible, but you are very strict about the phrase "along the way". You should never go backwards – you should always make progress towards your destination, even if its not the fastest way to San Jose.

We model this problem with a directed graph $G = (V, E)$ with positive real edge lengths $\ell : E \to \mathbb{R}_{>0}$. You have a vertex $s$ marking the start and a vertex $t$ marking the end. You are given a set $A \subseteq V$ of size $p = |A|$, which define the tourist attractions you would like to see. The goal is to compute the maximum number of attractions that you can visit along any walk $(v_0 = s), v_1, \ldots, (v_k = t)$ from $s$ to $t$ such that for $i = 1, \ldots, k$, the distance from $v_i$ to $t$ is (strictly) less than the distance from $v_{i-1}$ to $t$ (with respect to $\ell$). For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.67.** Implement `merge-sort` in $O(n \log n)$ time using only (immutable) stacks. Here the input consists of a stack of elements. The output is a stack of elements in sorted order (increasing from the top of the stack to the bottom.)

**Exercise K.68.** Let $A[1..n]$ be an array of $n$ integers and let $B[1..n]$ be an array of $n$ characters. We say that a subsequence $i_1, i_2, ..., i_k$ of $1, \ldots, n$ is *increasing in* $A$ if the corresponding subsequence of $A$, $A[i_1], \ldots, A[i_k]$, is (weakly) increasing (i.e., $A[i_i] \leq A[i_2] \leq \cdots \leq A[i_k]$). We say that a subsequence $i_1, i_2, \ldots, i_k$ of $1, \ldots, n$ is a *palindrome in* $B$ if the corresponding subsequence of $B$, $B[i_1], B[i_2], \ldots, B[i_k]$, forms a palindrome.

Consider the problem of computing the length of the longest subsequence of $1, \ldots, n$ that is both increasing in $A$ and a palindrome in $B$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.69.** Recall that binary search can find one element out of a sorted list of elements in $O(\log n)$ time. (Assuming the elements are arranged in, say, an array, so that we can access the $i$th element in constant time.) Here we will try to obtain a matching lower bound.

The general problem gives as input a list of $n$ elements $(x_1, \ldots, x_n)$ in sorted order and with constant time access for any index, as well as an additional element $k$ that acts as a key. For simplicity we may assume that $k$ is promised to be one of the elements. The goal is to identify an index $i$ such that $x_i = k$. We restrict ourselves to a comparison-only model where we may ask comparison queries such as "is $k < x_i$?", "is $k \leq x_i$?", "is $k = x_i$?", and so forth. Prove a lower bound of $\Omega(\log n)$ comparison queries (in the worst case) for any search algorithm that correctly finds $k$ in the sorted list $(x_1, \ldots, x_n)$.

**Exercise K.70. Evacuate the building!** Before Purdue can begin construction on its next glorious computer science building, at least two times the size of Lawson and with at most half as many restrooms, the architectural plan has to pass a series of safety regulations. One of the most important safety requirements is having an *evacuation plan* where in the event of an emergency, one can empty all the rooms to safe locations without overcrowding any of the hallways and other pathways, and risking a stampede. Here we model the problem of computing a safe evacuation plan (or declaring that none exists) as a graph problem, as follows.

The input consists of a directed graph $G = (V, E)$, two disjoint subsets of vertices $X, Y \subseteq V$, and integer edge labels $z : E \to \mathbb{N}$. $X$ represents the rooms that would need to be evacuated, and $Y$ represents the safe locations that the rooms in $X$ must be evacuated to. (One safe location can be the destination of any number of evacuation routes.) The edges represent hallways/pathways through the building, and each edge $e$ is annotated with an integer label $z(e)$ that signifies the number of evacuation paths it can accommodate. We define an *evacuation plan* as a set of paths from rooms in $X$ to safety locations in $Y$, with one path starting from each room in $X$. We say that an evacuation plan is *safe* if no hallway/pathway is used more than its declared limit $z(e)$.

The problem is to either compute a safe evacuation plan or declare that no safe evacuation plan exists. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.71.** Let $T$ be a tree on $n$ vertices. A *centroid* of $T$ is a vertex $v$ whose removal breaks $T$ into subtrees of at most $n/2$ vertices. A tree can have up to two centroids.

1. Give a linear time algorithm taking as input a tree and returning a centroid. Your proof of correctness doubles as a proof that every tree has a centroid.

Centroids can be used as a sort of "pivot" in divide-and-conquer algorithms over trees. Design and analyze an algorithm for each of the following problems, leveraging centroids and recursion.[33]

2. Design and analyze an algorithm that computes, for a given $k \in \mathbb{N}$, the number of pairs of vertices that are $k$ edges apart in $T$.

3. Let each edge by labeled by a fixed-length bit strength (of the same length). Assuming we can compute the XOR of two bit strings in $O(1)$ time, design and analyze an algorithm that, given a string $x$, computes the number of paths in $T$ where the XOR over the edges in the path equals $x$.

4. Under the same setting as the previous problem, design and analyze an algorithm that maximizes the XOR over its labels (where each bit string is interpreted as an integer encoded in binary).

For the following, let the edges of $T$ be labeled by "lengths" $\ell(e)$. The distance between two vertices is the sum of edge lengths over the unique path between the two vertices in $T$.

5. Design and analyze an algorithm that computes, for each vertex $v$, the sum of distances to all other vertices.

6. Design and analyze an algorithm that computes the diameter of $T$; i.e., the pair of vertices of maximum distance in $T$.

7. Define the *radius* of $T$ as the minimum, over all vertices $v \in T$, of the maximum distance from $v$ to any other vertex. The vertex $v$ minimizing the maximum distance is called the "1-median" Design and analyze an algorithm computing the radius of $T$ and the 1-median.

8. Design and analyze an algorithm that, given values $a, b \in \mathbb{R}$ with $a \leq b$, counts the number of pairs with distance in the interval $[a, b]$

9. Design and analyze an algorithm that counts the number of ordered pairs $(u, v)$ where the edge weights along the $(u, v)$-path are (strictly) monotonically increasing.

[33] A couple of these problems can be solved by other means; for pedagogical reasons, we still encourage you to solve them via centroids.

10. Design and analyze a data structure that, after preprocessing $T$, can answer "distance queries" that take as input two vertices $s, t$ and output the distance between $s$ and $t$.

11. Suppose that the edges have both lengths $\ell(e) \in \mathbb{R}$ and costs $c(e) \in \mathbb{R}$. Design and analyze an algorithm that, given $C, D \in \mathbb{R}$, counts the number of pairs of vertices $u, v$ with distance at most $D$ and where the cost of the $(u, v)$-path is at most $C$.

**Exercise K.72.** Let $G = (V, E)$ be a directed graph. We say an edge $(u, v)$ is *irreversible* if there is no walk from $v$ to $u$ in the graph. Design and analyze an algorithm (the faster the better) that computes all the irreversible edges in $G$.

**Exercise K.73.** Consider the line-breaking problem from Section 5.4. Suppose that you are given an additional input parameter $k$, which defines the maximum number of words allowed in a line. (You may assume $k \leq n$, and morally, $k \ll n$). Show how to modify the algorithm from Section 5.4 to give a faster running time (in terms of $n$ and $k$).

**Exercise K.74.** Consider the following special case of SAT, which we will call $k$-*occurrence-SAT* for a fixed parameter $k \in \mathbb{N}$. The input consists of a SAT formula $f(x_1, \ldots, x_n)$ in CNF such that every variable $x_i$ appears (as is, or negated) in at most $k$ clauses. The problem is to decide whether there is a satisfying assignment. For $k = 3$, either (a) design and analyze a polynomial time algorithm, or (b) show that a polynomial time algorithm for $k$-occurrence-SAT implies a polynomial time algorithm for (CNF-)SAT. [34]

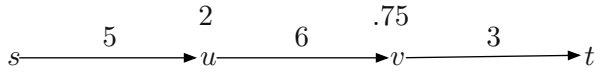
**Exercise K.75.** In various games like Mario Kart, not only are you trying to get from start to finish as fast as possible, but there are various items along the way that speed you up or slow you down. These items inspire the "multipliers" in the following variation of shortest paths.

Let $G = (V, E)$ be a DAG and $s, t \in V$. Let the edges have positive labels $\ell : E \to \mathbb{R}_{>0}$ representing the amount of time it takes to traverse the edge at unit speed. For each vertex $v \in V$, let $\alpha(v) > 0$ be a "speed multiplier". For example, if a

[34] As a warmup, it might be helpful to first consider the case $k = 5$. If you figure out 5-occurrence SAT, but don't figure out 3-occurrence SAT, we will give partial credit for a solution to 5-occurrence SAT.

vertex $v$ has multiplier $\alpha(v) = 2$, then your speed doubles as you pass through $v$; if $\alpha(v) = 1/2$, then your speed decreases by a factor of 2 as you pass through $v$.

You start at $s$ in a vehicle at unit speed. Your goal is to get to $t$ as fast as possible. Along the way your speed increases or decreases multiplicatively based on the speed multipliers $\alpha(v)$ of each vertex you traverse. (For simplicity we assume $\alpha(s) = \alpha(t) = 1$.)




For example, in the path from $s$ to $t$ above, the edges are labeled by the time at unit speed, and the vertices are labeled by the speed multipliers. Going from $s$ to $t$, the first edge is traversed at unit speed and takes 5 units of time, the second edge is traversed at 2x speed and takes 3 units of time, and the last edge is traversed at 1.5x speed (since $2 \times .75 = 1.5$) and takes 2 units of time, for 10 units of time total.

Consider the problem of computing the minimum amount of time needed to get from $s$ to $t$, taking into account the speed multipliers of each vertex. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

(Extra credit / food for thought: what if $G$ was a general graph?)

**Exercise K.76.** Prove linearity of expectation (Theorem 25.6).

**Exercise K.77.** Let $x_1, \ldots, x_n \in \mathbb{N}$. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.[35]

1. The *partition problem* asks if one can partition $x_1, \ldots, x_n$ into two parts such that the sums of each part are equal.

2. The 3-*partition problem* asks if one can partition $x_1, \ldots, x_n$ into 3 parts such that the sums of each part are all equal.

3. The *any-k-partition problem* asks one can partition $x_1, \ldots, x_n$ into $k$ parts, for any integer $k \geq 2$, such that the sums of each part are all equal.

4. The *almost-partition* problem asks if one can partition $x_1, \ldots, x_n$ into two parts such that the two sums of each part differ by at most 1.

[35] You can use the solution of one subproblem to solve another, as long as there's no circular dependencies overall.

5. [36]Let $n$ be even. The *perfect partition problem* asks if one can partition $x_1, \ldots, x_n$ into two parts such that
   (a) Each part has the same sum.
   (b) Each part contains the same number of $x_i$'s.

**Exercise K.78.** Prove or disprove: for any two random variables $X, Y$, and real-valued function $f(X, Y)$, we have

$$\mathop{\mathbf{E}}_{X}\Big[\mathop{\mathbf{E}}_{Y}[f(X,Y) \mid X]\Big] = \mathop{\mathbf{E}}_{Y}\Big[\mathop{\mathbf{E}}_{X}[f(X,Y) \mid Y]\Big].$$

**Exercise K.79.** This problem is meant to model the following scenario. Suppose you drive to campus but you are running late for the CS580 midterm, and in this extreme circumstance you are willing to speed up to 25% over the speed limit. However, you don't want to run too high a risk of getting a ticket, so you decide you are willing to speed for a limited number $k$ of streets. The goal is to get to the exam as fast as possible, going 25% over the speed limit for up to $k$ streets, and obeying the speed limit everywhere else.

Formally, the input consists of a directed graph $G = (V, E)$, positive edge weights $\ell : E \to \mathbb{R}_{>0}$, vertices $s, t \in V$, and an integer $k \in \mathbb{N}$. The *$k$-speeding distance* from $s$ to $t$ is defined as the minimum total length of any $(s, t)$-walk, except there the $k$ largest edges are decreased by a factor of $4/5$. For example, if $k = 3$, and you have an $(s, t)$-walk with edge lengths $6, 3, 4, 6, 9, 3, 6, 9, 7, 1$, then the total length including the $k$ "speedups" becomes

$$6 + 3 + 4 + 6 + \frac{4}{5} * 9 + 4 + 6 + \frac{4}{5} * 9 + \frac{4}{5} * 7 + 1 = 50.$$

Design and analyze an algorithm computing the $k$-speeding distance from $s$ to $t$. Your running time may depend on $k$. (You may assume $k < n$ since any shortest walk should be a path anyway).

**Exercise K.80.** Peter Venkman, Ray Stantz, and Egon Spengler were three professors at Columbia researching paranormal activity, broadly construed. After their first encounter with a ghost at the New York Public Library, the university dean dismisses the credibility of their paranormal-focused research and fires them. The trio

[36]IMO, this one is the trickiest.

responded by establishing "Ghostbusters", a paranormal investigation and elimination service operating out of a disused firehouse. They develop high-tech nuclear-powered equipment to capture and contain ghosts. Business was initially slow but after a ghost epidemic they suddenly need to scale up their operation. Peter, Ray and Egon need to build a ghostbuster army.

After placing some television ads, and thanks in part to their catchy theme song, they were able to assemble $n$ recruits to their ghostbuster army. Now, to be in the ghostbuster army, you can't be afraid of no ghosts, and you definitely can't be afraid of the dark. To test the latter, Peter, Ray and Egon put all $n$ recruits in a room and turned off the lights. One of the recruits screamed. They turned the lights back on. Apparently one of the recruits is afraid of the dark.


Peter, Egon, and Ray

The ghostbusters need to figure out who in the army is afraid of the dark. Nobody wants to admit they are afraid of the dark. But we can test if someone is afraid of the dark by putting them in a room, and turning off the lights. A person screams if and only if they are afraid of the dark.

Of course we can identify the scaredy-cat by testing each recruit one by one. This can take a long time. The ghostbusters have other options: they can place a bunch of recruits in the dark, and by turning off the lights, figure out if the scaredy-cat is among them.

Let's say that it takes "one round" to put a set of recruits in the room, turn off the lights, and see if any of the recruits are scared of the dark. Design and analyze an algorithm that identifies a recruit that's scared of the dark in the minimum number of rounds.

**Exercise K.81.** In fair redistricting, the goal is to divide up a population into geographic regions so that all the local elections are as close as possible. This is like the opposite of gerrymandering; having close elections maximizes the influence of each constituent's vote.

For simplicity we consider the following 1-dimensional formulation of the problem. Let $A[1..n]$ be a sequence of $n$ (positive or negative) integers. Each index $i \in [n]$ represents a neighborhood block in a sequence of $n$ blocks. We have two political parties, the Apples and the Oranges. For each index $i$, $A[i] \in \mathbb{Z}$ represents the number

of people in block $i$ that have registered for the Apple party, minus the number of people in block $i$ that have registered for the Orange party; i.e.,

$$A[i] = \begin{pmatrix} \text{\# Apples} \\ \text{in block } i \end{pmatrix} - \begin{pmatrix} \text{\# Oranges} \\ \text{in block } i \end{pmatrix}.$$

A *district* is defined as a consecutive interval $[i..j]$ of blocks, where $1 \leq i \leq j \leq n$. The *disparity* of a district $[i..j]$ is the absolute difference between the number of Apples and number of Oranges in the district:

$$(\text{disparity of } [i..j]) = \left| \begin{pmatrix} \text{\# Apples} \\ \text{in blocks } i..j \end{pmatrix} - \begin{pmatrix} \text{\# Oranges} \\ \text{in blocks } i..j \end{pmatrix} \right| = \left| \sum_{k=i}^{j} A[k] \right|.$$

Given $A[1..n]$ and $L \in [n]$, the goal is compute the minimum total disparity,

$$\sum_{k=1}^{L} (\text{disparity of } A[i_k..j_k])$$

over all partitions of the sequence blocks $[1..n]$ into $L$ (nonempty) districts $[i_1..j_1], \ldots, [i_L..j_L]$. ("Partition" means that the districts are disjoint and cover all of the $n$ blocks.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.82.** Theorem 13.3 proved that a polynomial time algorithm for 3-coloring implies a polynomial time algorithm for SAT. Prove that for any $k > 3$, a polynomial time algorithm for $k$-coloring implies a polynomial algorithm for SAT.[37]

**Exercise K.83.** Let $G = (V, E)$ be a directed graph with integer edge capacities $c : E \to \mathbb{N}$. Let each edge be assigned a color `red`, `white`, or `blue`. Recall that an *American walk* is a walk where the edges alternate `red`, `white`, `blue`, `red`, `white`, `blue`, ... Here the first color may be any color; all that matters is that we keep cycling through the colors patriotically. Given $s, t \in V$, consider the problem of computing the size of a maximum packing of American $(s, t)$-walks.

(Here the definitions are just like for $(s, t)$-paths. A "packing of walks" is a collection of walks that, for each edge $e$, uses the edge $e$ at most as many times as its capacity. The "size of a packing" refers to the number of walks.)

[37] It might be helpful to focus on $k = 4$ to develop your ideas.

**Exercise K.84.** Given a set of numbers $x_1, \ldots, x_n$, the *prefix sums* are the $n$ numbers,

$$x_1,\ x_1 + x_2,\ x_1 + x_2 + x_3,\ \ldots,\ x_1 + \cdots + x_n.$$

Of course the prefix sums can be computed in $O(n)$ time, one prefix sum at a time. What about in *parallel*; e.g., on a GPU?

Here we consider a *parallel* computation model, called *PRAM*. To motivate this model, imagine one had arbitrarily many processors, with shared memory, trying to solve a common problem. Even with tons of computing, there may still be sequential bottlenecks — some steps have to be executed before others — and issues of synchronization.

In the PRAM model, the processors have access to shared memory. To simplify the synchronization, we restrict our model to work in *rounds*. In each round, each processor reads a constant amount of information from memory, does a constant amount of work, and writes to a constant number of locations in memory. We will restrict ourselves to algorithms where in each round, different processors always write to different locations, so we can avoid issues of contention resolution.

The number of rounds needed to complete the task is the *parallel running time*. The total number of operations, over all processes, is called the *total amount of work*. The ideal parallel algorithm has total work similar to the best known sequential algorithm, but has a much faster parallel running time.

For example, here is a first attempt at a parallel divide-and-conquer algorithm for prefix sums. We first define our recursive spec:

> `prefix-sum(`$A[1..n]$`)`: Given an array of numbers $A[1..n]$, returns an array of the prefix sums of $A$.

Our first implementation divides $A$ into equal-size subarrays and recursively computes the prefix sums on each. Observe that the prefix sums of the second half are missing the sum from the first half. We fix this by taking the last prefix sum of the first half and adding it to each prefix sum in the second half. See Fig. K.1 for pseudocode.

Let $T(n)$ denote the parallel running time on an input of size $n$. We have

$$\begin{aligned} T(0) &= T(1) = 1 \\ T(n) &= T(n/2) + O(1) \qquad\qquad \text{for } n > 1. \end{aligned}$$

Here we observed that although parallel prefix sum has two recursive calls, they are made in parallel. This is the same recurrence as (sequential) binary search, and implies a $O(\log n)$ parallel running time.

`prefix-sum(`$A[1..n]$`)`

*/* Given an input array of numbers $A[1..n]$, returns an array of the prefix sums of $A$. */*

1. If $n \leq 1$ then return $A$.
2. Let $k = \lceil n/2 \rceil$.
3. $B[1..k] \leftarrow$ `prefix-sum(`$A[1..k]$`)` and $C[1..n-k] \leftarrow$ `prefix-sum(`$A[k+1..n]$`)` (in parallel).
4. $D[i] \leftarrow B[k] + C[i]$ for $i = 1, \ldots, n-k$. *// $O(n)$ work in $O(1)$ rounds*
5. Return the concatenation of $B[1..k]$ and $D[1..n-k]$. *// $O(n)$ work in $O(1)$ rounds*

Figure K.1: A parallel prefix sum algorithm with $O(\log n)$ time and $O(n \log n)$ work.

Now we compute the total amount of work. This is the same as the running time if the `parallel-prefix-sum` algorithm was simulated sequentially. Letting $W(n)$ be the total amount of work on an input of size $n$, we have

$$\begin{aligned} W(0) &= W(1) = 1 \\ W(n) &= 2W(n/2) + n/2 \qquad\qquad \text{for } n > 1. \end{aligned}$$

This is the same recurrence as merge sort and other algorithms we've seen, and gives $O(n \log n)$ total work.

This parallel algorithm is very fast, but has a $O(\log n)$-factor more work than the obvious sequential algorithm. The goal of this exercise is to develop a (recursive) parallel divide-and-conquer algorithm that is just as fast — $O(\log n)$ parallel time — but has $O(n)$ total work. In addition to designing the algorithm, you should analyze the running time and the work similar to how we did above. (You are welcome to use the same recursive spec, but don't forget to write it out.)

It may be fun to try to figure this out without any hints. (Maybe you'll come up with a new approach!) But if you want a little help, then there are some guiding questions in the following footnote.[38]

[38]Hint: Suppose you had the prefix sums $y_i = x_1 + x_2 + \cdots + x_i$ for all *even* $i$. Can you get the prefix sums for all *odd* $i$ in $O(1)$ rounds and $O(n)$ work?

And then: how do you get the prefix sums for even $i$ efficiently? Of course you could compute all the prefix sums, and then pick out the even indices $i$. But that's not going to be help us speed up

*Remark* K.1. To generalize beyond summing numbers, observe that the only property of addition we used was the associative property:

$$a + (b + c) = (a + b) + c.$$

So with essentially the same algorithm, one can compute generalized prefix sums for any associative binary operator in $O(\log n)$ rounds and $O(n)$ work.

**Exercise K.85.** In the playoff elimination problem (Section 19.1.4), we asked whether a particular team was mathematically eliminated from making the playoffs. This allows the home team to win all of its remaining games, and other teams to lose all of their games, which may be unrealistic when there are many games left. Suppose we assumed that:

1. No team will win more than 90% of its remaining games, rounded up. (I.e., if a team has $k$ games left, it can win at most $\lceil .9k \rceil$ of them.)
2. Every team will win at least 10% of its remaining games, rounded down. (I.e., if a team has $k$ games left, it must win at least $\lfloor .1k \rfloor$ of them.)

Consider the problem of deciding whether the home team can make the playoffs under the additional restrictions listed above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.86.** In Corollary 4.3, we saw that FFT can multiply two degree-$n$ polynomials in $O(n \log n)$ time. Now consider the problem of multiplying $n$ degree-1 polynomials. The input consists of $n$ linear polynomials

$$a_1 + b_1 x, , \, a_2 + b_2 x, \, \ldots, \, a_n + b_n x,$$

and the output is the coefficients of their product, a degree $n$ polynomial. Design and analyze an algorithm, as fast as possible, for this problem.

**Exercise K.87.** Extend the approximation algorithm for set cover to positive costs. For each set, there is a positive cost $c_j > 0$. The goal is to compute the minimum cost collection of sets that covers all the points.

computing all the prefix sums. Can you get the prefix sums for even $i$ by creating another instance of the prefix sums problem with significantly fewer numbers?

**Exercise K.88.** At the end of Section 11.1, we discussed how to use an additional table to also extract the subset sum solution. Here we develop an alternative approach that was briefly alluded to in Section 11.1. It is not as efficient as building a secondary table, but the technique is more general and of independent interest.

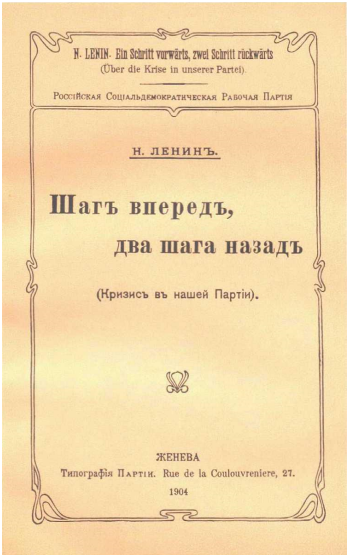
Suppose one had a black box algorithm for deciding subset sum problems. That is, given an instance of subset sum described by $n$ integers $x_1, \ldots, x_n \in \mathbb{N}$ and an additional target value $T$, the algorithm returns `true` if there exists a feasible solution, and otherwise `false`. Show that one can use this decision algorithm as a subroutine to obtain an efficient constructive algorithm for subset sum.

**Exercise K.89.** Let $G = (V, E)$ be a directed graph. For $s, t \in V$, let $\lambda(s, t)$ denote the size of the minimum $(s, t)$-cut. Prove the following "triangle inequality" for cuts:

$$\lambda(a, b) \geq \min\{\lambda(a, c), \lambda(c, b)\}$$

for any three vertices $a, b, c \in V$.

**Exercise K.90.** *One Step Forward, Two Steps Back: The Crisis in Our Party* (Russian, romanized: *Shag vperyod, dva shaga nazad (Krizis v nashey partii)*) is a work written by Vladimir Lenin and published on May 6/19, 1904. In it Lenin defends his role in the 2nd Congress of the Russian Social Democratic Labour Party, held in Brussels and London from July 30 to August 23, 1903. Lenin examines the circumstances that resulted in a split within the party between a Bolshevik ("majority") faction, led by himself, and a Menshevik ("minority") faction, led by Julius Martov.




Written in 1904 in response to controversies within the Social Democratic Labour Party's Second Congress regarding the status of party membership and organization, Lenin frames this conflicting factionalism within the Party in terms of dialectics. According to Lenin, there are two conflicting factions within the party: "the revolutionaries", which consists of the majority of party members (the Bolsheviks) and "the opportunists", which are the minority (the Mensheviks).
Rosa Luxemburg, a Marxist living in Germany at the time, responded to the article in *Organizational Questions of the Russian Social Democracy* (1904). She criticized Lenin's attitude towards democratic centralism and wrote about the role of "spontaneity" among the working class. However, different authors make varying claims about her precise attitude towards Lenin, the Bolshevik faction and the revolutionary situation in Russia.

Let $G = (V, E)$ be a directed graph. A *Lenin walk* is defined as a sequence of vertices $v_1, ..., v_k$ such that for each consecutive pair of vertices $(v_i, v_{i+1})$:

- If $i \mod 3 = 1$, then $(v_i, v_{i+1}) \in E$.
- If $i \mod 3 \in \{0, 2\}$, $(v_{i+1}, v_i) \in E$.

If we call each pair $(v_i, v_{i+1})$ in the Lenin walk a "step", then a Lenin walk alternates between one step going forwards along an edge and two steps each going "backwards", in the opposite direction of an edge.

Given $G$ and $s, t \in V$, consider the problem of computing the infimum number of vertices along any Lenin walk from $s$ to $t$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.91.** Show that any Boolean formula with $n$ variables and total size $m$ can be converted to a Boolean formula with $n$ variables and total size $O(m)$ such that every clause contains at most two sub-clauses or variables. That is, in the tree representation of the formula, each $\wedge$ and $\vee$ would have two children. For example, the formula for XOR, $f(x_1, x_2) = (\bar{x}_1 \wedge x_2) \vee (x_1 \wedge \bar{x}_2)$, is in this form.

**Exercise K.92.** A Boolean formula is in *disjunctive normal form (DNF)* if it is a disjunction of conjunctions. For example, the formula

$$f(x, y, z) = (x \wedge y) \vee (\bar{x} \wedge z),$$

which models "if $x$ then $y$ else $z$", is in disjunctive normal form. Note that by the distributive property,

$$(a \vee b) \wedge (c \vee d) = (a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d),$$

so any CNF formula can be converted to a DNF formula. Let us define *DNF-SAT* as the problem of deciding whether a boolean formula in DNF is satisfiable.

1. Is there a polynomial time algorithm for DNF SAT? If so, describe and analyze such an algorithm.

2. Would a polynomial time algorithm for DNF SAT imply a polynomial time algorithm for CNF SAT? Explain your reasoning.

**Exercise K.93.** Show that the construction given in Section 27.2 is indeed a universal hash function, using the steps listed below.

To recall the construction, we randomly construct a function $h : [n] \to [k]$ as follows. First, let $p$ be any prime number $> n$. Draw $a \in \{1, \ldots, p-1\}$ uniformly at random, and draw $b \in \{0, \ldots, p-1\}$ uniformly at random. We define a function $h(x)$ by

$$h(x) = ((ax + b) \mod p) \mod k.$$

1. Let $x_1, x_2 \in [n]$ with $x_1 \neq x_2$, and let $c_1, c_2 \in \{0, \ldots, p-1\}$ with $c_1 \neq c_2$. Show that the system of equations

$$\begin{aligned} ax_1 + b &= c_1 \mod p \\ ax_2 + b &= c_2 \mod p \end{aligned}$$

uniquely determines $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$.[39]

- Step 1 implies that the map $(a, b) \mapsto (ax_1 + b \mod p, ax_2 + b \mod p)$ is a bijection between $\{1, \ldots, p-1\} \times \{0, \ldots, p-1\}$ and $\{(c_1, c_2) \in \{0, \ldots, p-1\} : c_1 \neq c_2\}$.

[39] Here it is helpful to know that division is well-defined on the set of integers modullo $p$ when $p$ is prime. More precisely, "$a/b$" is defined as the unique integer $c$ such that $bc = a$.

2. Let $x_1, x_2 \in [n]$ with $x_1 \neq x_2$, and let $c_1, c_2 \in \{0, \ldots, p-1\}$ with $c_1 \neq c_2$. Show that
$$\mathbf{P}[ax_1 + b = c_1 \mod p,\ ax_2 + b = c_2 \mod p] = \frac{1}{p(p-1)}.$$
(Here the randomness is over the uniformly random choices of $a$ and $b$.)

3. Fix $x_1, x_2 \in [n]$ with $x_1 \neq x_2$, and $c_1 \in \{0, \ldots, p-1\}$. Show that
$$\sum_{\substack{c_2 \in \{1,\ldots,p\} \\ c_2 \neq c_1 \mod p \\ c_1 = c_2 \mod k}} \mathbf{P}[ax_1 + b = c_1 \mod p,\ ax_2 + b = c_2 \mod p] \leq \frac{1}{pk}.$$

- The LHS represents $\mathbf{P}[ax_1 + b = c_1 \mod p \text{ and } h(x_2) = h(x_1)]$.[40][41]

4. Finally, show that $\mathbf{P}[h(x_1) = h(x_2)] \leq \frac{1}{k}$.

**Exercise K.94.** Consider the particular case of hash tables with chaining with $k = n$ and an *ideal* hash function $h : [n] \to [n]$. Let $A[1..n]$ be the cells of the hash table.

1. Consider a particular array slot $A[i]$. Show that for $\ell \in \mathbb{N}$, the probability that $A[i]$ has $\geq \ell$ items hashed to it is
$$\mathbf{P}[\text{at least } \ell \text{ items being hashed to } A[i]] \leq \frac{1}{\ell!}.$$

2. Show that, with probability of error $\leq 1/n^2$, the maximum length is at most $O(\log(n)/\log\log n)$.[42]

**Exercise K.95.** Let $G = (V, E)$ be a directed graph with positive edge weights $\ell : E \to \mathbb{R}_{>0}$. Let $s, t \in V$ be distinct vertices such that $s$ can reach $t$. Recall that the distance from $s$ to $t$ can be computed in $O(m + n \log n)$ time, and that there may be many shortest $(s,t)$-walks. We say a vertex $v$ is *on the way* from $s$ to $t$ if $v$ is on a shortest $(s,t)$-walk. (Yes, $s$ and $t$ themselves are both on the way from $s$ to $t$. And yes, with positive edge weights, all shortest walks are shortest paths.)

Consider the problem of computing the set of all vertices $v$ on the way from $s$ to $t$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

[40] Here we note that for $x_1 \neq x_2$, $ax_1 + b = ax_2 + b_2 \mod p$ iff $a = 0$.

[41] *Hint:* You may want to show that the number of values $c_2 \in [p]$ such that $c_1 = c_2 \mod k$ is $\leq \frac{p-1}{n}$.

[42] The simple lower bound of $\ell! \geq (\ell/2)^{\ell/2}$ may be helpful. It is implicit in the $O(\cdots)$ notation that your bound need only hold for $n$ sufficiently large.

**Exercise K.96.** While we showed that Hamiltonian path is unlikely to permit polynomial time algorithms, perhaps it is still interesting to try to develop faster exponential time algorithms. The simplest approach enumerates all permutations of the vertices and checks to see if they describe a Hamiltonian cycle; this takes $O(n! \operatorname{poly}(n))$ time. Can one do better? Design and analyze an algorithm that computes a Hamiltonian path in $O(2^n \operatorname{poly}(n))$ time. (The smaller the $\operatorname{poly}(n)$ term the better, but the key point is to reduce $n!$ to $2^n$.)

**Exercise K.97.** Suppose one had access to a polynomial time algorithm that solves the decision version of Boolean satisfiability (outputting `true` or `false` depending on whether there *exists* a satisfying assignment.). Show how to use this algorithm to obtain a polynomial time algorithm for the constructive version of Boolean satisfiability, outputting a satisfying assignment if one exists.

**Exercise K.98.** Suppose you had a polynomial time algorithm for the decision version of the Hamiltonian path problem. That is, given a directed graph $G$, the algorithm returns `true` if $G$ has a Hamiltonian path, and `false` if not. Consider the problem of constructing a Hamiltonian path: that is, if $G$ has a Hamiltonian path, then return a Hamiltonian path. Show how to use the polynomial time algorithm for the decision problem to obtain a polynomial time algorithm for Hamiltonian path.

**Exercise K.99.** Let $I_1, \ldots, I_n$ be a collection of $n$ intervals on the real line, and let $p_1, \ldots, p_m \in \mathbb{R}$ be a collection of $m$ points on the line. Let each interval $I_j$ be assigned a positive weight $w(I_j) > 0$.

An interval $I_j$ *covers* a point $p_i$ if $p_i \in I_j$. A collection of intervals $I_{j_1}, \ldots, I_{j_k}$ is an *interval cover* of $p_1, \ldots, p_m$ if every point $p_i$ is covered by at least one interval $I_{j_h}$ in the collection. We assume that the collection of all the intervals $I_1, \ldots, I_n$ is an interval cover of $p_1, \ldots, p_m$. We also assume that all the points are distinct.

Consider the problem of computing the minimum weight of any interval cover of $p_1, \ldots, p_m$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.100.** Let $X[1..n]$ be a sequence of integers in the range $[-k, k]$. A *consecutive sum* is defined as the sum of a consecutive subsequence of the form $X[i..j]$. Consider the problem of computing the number of *distinct* consecutive sums in $X$. Of course we can compute this in $O(n^2 \log n)$ or $O(n^2 + nk)$ time, by generating all prefix sums $X[1..i]$, and then marking all consecutive sums $X[i..j] = X[1..j] - X[1..i]$

in a lookup table (implemented as a balanced tree or as an $n \times k$ array). But suppose $k$ is much smaller than $n$. Design and analyze an algorithm computing the number of distinct consecutive sums in $O(n \operatorname{poly}(k, \log(n)))$ time. (Of course, the smaller the $\operatorname{poly}(k, \log n)$ term, the better.)

**Exercise K.101.** Consider the following variation of subset sum. The input consists of $n$ (positive) integers $x_1, \ldots, x_n \in \mathbb{N}$, an integer $T \in \mathbb{N}$, and an integer $k \in \{1, \ldots, n\}$. The problem is to decide if there is a subset of $x_1, \ldots, x_n$ of cardinality $k$ that sums to $T$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.102.** This exercise develops a $O((m^2 + mn \log(n)) \log(\lambda))$-time algorithm for maximum $(s,t)$-flow and builds on ideas from exercise K.12.

1. Prove the following: Given any $(s,t)$-flow problem with max flow value $\lambda > 0$, there exists an $(s,t)$-path where the minimum capacity edge is at least $\lambda/m$.
2. Describe a $O(m + n \log(n))$-time algorithm to find the path described above.[43]
3. Based on the two parts above, design and analyze an $(s,t)$-max flow algorithm that runs in $O(m(m + n \log(n)) \log(\lambda))$ time for integer capacities, where $\lambda$ denotes the value of the maximum flow.[44][45] (The algorithm does not know the true value of $\lambda$ *a priori*.)

**Exercise K.103.** Consider the problem of computing the second smallest weight spanning tree of a weighted graph $G$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.104.** Let $k \in \mathbb{N}$ be fixed. Prove that for $n \geq k$, there exists a $O(n)$-size boolean formula for the $n$-variable function

$$f(x_1, \ldots, x_n) = \begin{cases} \texttt{true} & \text{if at least } k \text{ of the } x_i\text{'s are } \texttt{true} \\ \texttt{false} & \text{otherwise.} \end{cases}$$

[43] $O(m \log n)$ time is a little easier and this running time would still get partial credit. Even if the $O(m + n \log(n))$-running time eludes you, you can assume it as a black box for the next part.

[44] This is polynomial with respect to the bit complexity of the input.

[45] A possibly helpful bit of math: for (small) $\epsilon > 0$, $\log_{1+\epsilon}(x) \leq O(\log(x)/\epsilon)$ is a good approximation (which you may want to verify for yourself).

**Exercise K.105.** Show that any CNF with $n$ variables and total size $m$ can be converted into a 3-SAT problem with $O(m)$ clauses and $O(m+n)$ variables.

**Exercise K.106.** Let $G = (V, E)$ be an undirected graph with positive edge weights $w : E \to \mathbb{R}_{>0}$. We define the *weighted density* of a graph as the total edge weight divided by the number of vertices. Consider the problem of computing the weighted densest subgraph of $G$, assuming the edge weights are all integral and between 1 and poly$(n)$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.107.** Two points $(a, b), (c, d) \in \mathbb{R}^2$ are *comparable* if either (a) $a \leq c$ and $b \leq d$, or (b) $c \leq a$ and $d \leq b$.

Given a set of $n$ points $(a_1, b_1), \ldots, (a_n, b_n) \in \mathbb{R}^2$, consider the problem of computing the number of pairs of points that are comparable. You may assume for simplicity that all points are distinct. Design and analyze an algorithm (as fast as possible) for counting the number of comparable pairs of points.[46]

**Exercise K.108.** Let $G = (V, E)$ be a directed graph. For distinct vertices $s, t \in V$, an $(s, t)$-*vertex cut* is a set $X \subseteq V \setminus \{s, t\}$ such that every path from $s$ to $t$ passes through some vertex in $X$. Two paths are *internally vertex-disjoint* if they share no vertices except possibly $s$ and $t$. *Menger's theorem* states that the maximum number of internally vertex-disjoint $(s, t)$-paths equals the minimum size of an $(s, t)$-cut. Our goal is to prove Menger's theorem.

1. For disjoint sets $S, T \subseteq V$, an $(S, T)$-*path* is a directed path starting from $S$ and ending in $T$. Two $(S, T)$-paths are *vertex-disjoint* if they share no vertices, including their endpoints. An $(S, T)$-*vertex cut* is a set of vertices $X \subseteq V$ intersecting every $(S, T)$-path. Prove that if there exist $k$ vertex-disjoint $(S, T)$ paths, then every $(S, T)$-vertex cut has size at least $k$.

2. Prove the converse: if every $(S, T)$-vertex cut has size at least $k$, then there exist $k$ vertex-disjoint $(S, T)$-paths.

3. Now prove Menger's theorem (as stated above).

[46] *Hint:* As a warmup, try to instead compute the number of *incomparable* pairs of points. (A pair of points is incomparable if is not comparable. E.g., $(x_1, y_1)$ and $(x_2, y_2)$ are incomparable if $x_1 < x_2$ and $y_1 > y_2$.) You may want to do exercise K.158 before this one.

4. There is also an edge version: an *$s$-$t$ edge cut* is a set of edges whose removal disconnects $s$ from $t$, and two paths are *edge-disjoint* if they share no edges. Prove that the maximum number of edge-disjoint $(s,t)$-paths equals the minimum size of any $(s,t)$-edge cut.

**Exercise K.109.** Consider the following variation of subset-sum called *Kobe subset sum*. The input is similar to normal subset sum, with $n$ numbers $x_1, \ldots, x_n \in \mathbb{N}$ and $T \in \mathbb{N}$, except we are promised that each number $x_i$ lies in the range $\{8, \ldots, 24\}$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.110.** Show that any boolean circuit with $n$ gates and $m$ edges can be converted into a boolean formula in CNF with $O(n)$ variables and total size $O(m)$.

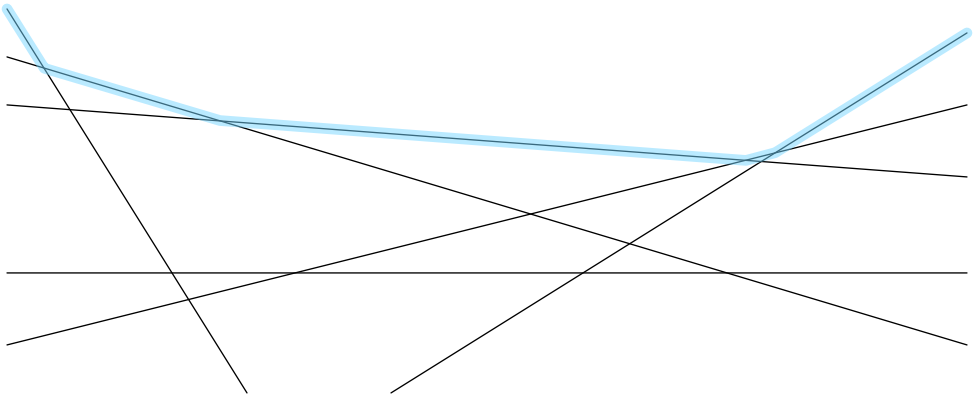
**Exercise K.111.** Here we consider a special case of SAT that we call *ordered-$k$-overlapping-3-SAT* for a fixed parameter $k$. We take as input a 3-CNF formula $f(x_1, \ldots, x_n)$ with $m$ clauses listed in order as to satisfy the following property. For every index $i$, there are at most $k$ variables $x_j$ with index $j < i$ that share a clause with the any variable $x_j$ with index $j \geq i$. For this special case of 3-SAT, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.112.** Let $L$ be a set of $n$ lines of the plane, where the $i$th line is of the form $y_i(x) = a_i x + b_i$ for $a_i, b_i \in \mathbb{R}$. The *upper envelope* is the function

$$U_L(x) = \max_{i \in [n]} y_i(x).$$

Observe that $U_L$ is piecewise-linear and convex. We represent the upper envelope as a list of *breakpoints* $(x_1, \ell_1), (x_2, \ell_2), \ldots, (x_m, \ell_m)$ where $x_1 < x_2 < \cdots < x_m$ are the $x$-coordinates where the topmost line changes, and $\ell_i$ is the line achieving the maximum for $x \in [x_i, x_{i+1})$.[47]

[47] You should also specify the line on top for $x < x_1$; handle this however you like.



For example, given three lines $\ell_1 \colon y = -x + 4$, $\ell_2 \colon y = 2$, and $\ell_3 \colon y = x$, the upper envelope is: $\ell_1$ for $x < 1$, then $\ell_2$ for $x \in [1, 2]$, then $\ell_3$ for $x > 2$.

Design and analyze a recursive algorithm, as fast as possible, computing the upper envelope of $n$ lines in the plane.[48]

*Remark* K.2. The upper envelope appears in computational geometry (convex hulls, linear programming) and in the "convex hull trick" for optimizing certain dynamic programs.

**Exercise K.113.** Let $A$ and $B$ be two events. Prove that the following three identities are all equivalent:

$$\mathbf{P}[A \mid B] = \mathbf{P}[A], \qquad \mathbf{P}[B \mid A] = \mathbf{P}[B], \qquad \mathbf{P}[A, B] = \mathbf{P}[A]\,\mathbf{P}[B].$$

(That is, if $A$ and $B$ satisfies any one of the identities above, then it automatically satisfies the other two.)

**Exercise K.114.** The defining characteristic of LPs is that the objective and all linear constraints are given by linear functions. It is natural to generalize this notion and consider mathematical programs where the objective and linear constraints are all given by low-degree polynomials; say, bounded by a degree $d$. Let us call these "degree $d$ polynomial programs". Linear programs are degree 1 polynomial programs.

Prove that degree $d$ polynomial programs are NP-Hard to solve for $d \geq 3$. To this end, pick a suitable NP-Hard problem, and design a degree 3 polynomial program that can be rounded to a discrete solution without any loss.[49]

[48] *Hint:* Of course $n$ lines induces $\binom{n}{2}$ points of intersection, but how many breakpoints can the upper envelope have?

[49] Better yet, prove the same for $d \geq 2$.

**Exercise K.115.** Let $k \geq 3$. Recall that the $k$-SAT problem is the special case of SAT where $f(x_1, \ldots, x_n)$ is a CNF with exactly $k$ distinct variables per clause.

1. Show that a polynomial time algorithm for $k$-SAT implies a polynomial time algorithm for $(k+1)$-SAT.
2. Show that a polynomial time algorithm for $(k+1)$-SAT implies a polynomial time algorithm for $k$-sat.

**Exercise K.116.** Consider the following recursive algorithm for exploring a directed graph with $n$ vertices and $m$ edges.

`edge-search(v)`

1. for each edge $e = (v \to w)$ leaving $v$
   A. if $e$ is unmarked
      1. mark $e$
      2. `edge-search(w)`

Analyze `edge-search` and give an upper bound (as tight as possible) on the running time, as a function of $m$ and $n$.

**Exercise K.117.** Recall the project selection problem in Section 19.1.3. Suppose we relax the problem so that the "precedence requirements" are no longer required to be acyclic. We instead interpret each set $R(p)$ as "co-requisites": if we take project $p$, then we also have to take on project $q$ for every $q \in R(p)$. (That is, we remove the emphasis that every $q \in R(p)$ should be executed "before" $p$.) For this generalization of project selection, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.118.** Recall the array-backed lists from Section 22.3, where we showed how to support `append` in $O(1)$ amortized time via the "doubling trick". As discussed at the end of Section 22.3, one may also want to include a `remove` operation that removes the last item in the list. To implement this, one could just unallocate the corresponding spot in memory, and decrease the counter tracking the size of our list. However, after many such removals, we may end up with an array that is mostly empty and much larger then the number of items in the list. It would be desirable to maintain the array to be within a constant factor. Here we will develop an analog of the doubling trick to facilitate deletions.

1. A first approach to addressing deletions might be as follows.

   > *After removing the element from the array and decreasing the size counter, if the size of the list is now less than half of the capacity of the array, allocate a new array of half the capacity and copy the elements of the list over.*

   Unfortunately, this approach will not run in $O(1)$ amortized time. To prove this, devise a sequence of $n$ `append` / `remove` operations (for $n$ arbitrarily larger) on which the data structure would take $\Omega(n^2)$ time.

2. While the approach above didn't quite work, a slight modification of it will. Implement `remove` so that `append` and `remove` both take $O(1)$ amortized time. (Analyze these operations as well.) Your implementation should still follow the general format of allocating a new array and copying all the elements at certain points in time.

**Exercise K.119.**
1. Consider the Hamiltonian path problem in *undirected* graphs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. Now consider the Hamiltonian *cycle* problem in undirected graphs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.120.** Let $G = (V, E)$ be a undirected graph with distinct edge weights and let $T$ be the MST. Suppose the weight of a single edge $e \in E$ is altered. The edge $e$ may or may not be in $T$, and the weight may have been increased or decreased. Describe and analyze an algorithm that fixes the MST.[50]

**Exercise K.121.** Recall that the $k$-coloring problem is to decide if there is a vertex coloring with at most $k$-colors. The subset $k$-coloring problem also specifies for each vertex $v$ a subset $S_v \subseteq [k]$, and the $k$-coloring is restricted to have each vertex $v$ colored by a color in $S_v$. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

1. The $k$-coloring problem in trees.

[50] Of course one could build an entirely new MST from scratch. Can one do better?

2. The $k$-coloring problem for intervals.
3. The subset $k$-coloring problem in trees, for fixed $k$.
4. The subset $k$-coloring problem for intervals, for fixed $k$.

**Exercise K.122.** [51]Let $G = (V, E)$ be an undirected graph with $m$ edges and $n$ vertices, and $k \in \mathbb{N}$. Recall that a *vertex $k$-coloring* (a.k.a. a *graph $k$-coloring*) is an assignment of colors (out of $\{1, \dots, k\}$) to each $v \in V$ so that no two adjacent vertices have the same color. When such an assignment exists, $G$ is said to be *$k$-colorable*. Now suppose that for each color $i = 1, \dots, k$, and each vertex $v \in V$, there is a positive *assignment cost* $c(i, v) > 0$ for assigning color $i$ to vertex $v$.

The following problem is called the *minimum cost graph coloring (MCGC)* problem. Given $G$ and $k$ as input, where $k \geq 3$, the goal is to find the minimum total assignment cost of any vertex $k$-coloring of $V$ (which is $+\infty$ if no such coloring exists).[52] We consider the problem both for general graphs and trees.

1. (5 points) Suppose $G = (V, E)$ is a graph with $m$ edges and $n$ vertices. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.
2. (5 points) Suppose $G = (V, E)$ is a tree with $n$ vertices. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.123.** Given integer values $b \in \mathbb{N}^V$, a *$b$-matching* is a set of edges $M \subseteq E$ where every vertex $v$ is incident to at most $b(v)$ edges in $M$. Consider the problem of computing the largest $b$-matching in a bipartite graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.124.** In CIPs we allowed each $x_j$ to be as large as we want. Suppose we added the constraint $x_j \leq 1$ for all $j$. Would the randomized rounding algorithm from Section 26.3 still obtain a $(1 - 1/e)$-approximation ratio? Why or why not?

[51]You may assume as fact that a polynomial time algorithm for $k$-coloring implies a polynomial time algorithm for SAT; this fact is otherwise left to the reader as exercise K.82.

[52]Technically this would be the "infimum" of all assignment costs of (valid) vertex $k$-colorings.

**Exercise K.125.** Let $A$ and $B$ two events. Prove or disprove that $A$ and $B$ are independent iff $\bar{A}$ and $\bar{B}$ are independent.

**Exercise K.126.** Show that any boolean formula with $n$ variables and total size $m$ can be converted to a boolean formula in CNF with $O(m+n)$ variables, $O(m)$ clauses, and total size $O(m)$.[53][54]

**Exercise K.127.** Given a 2-SAT formula, we say that a clause $C$ is "pure" if either both literals are positive (e.g., $(x_1 \vee x_2)$) or both literals are negative. Otherwise we say $C$ is "mixed". We say that a 2-SAT formula $f$ is pure if every clause is pure.

For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

1. Pure 2-SAT: Given a pure 2-SAT formula decide if there is a satisfying assignment.
2. Max pure 2-SAT: Given a pure 2-SAT formula, compute the maximum number of clauses that can satisfied by any assignment.

**Exercise K.128.** Let $T = (V, E)$ by a rooted binary tree. Recall that a *full* binary tree is a binary tree where every non-leaf node has exactly. Design and analyze an algorithm that computes the maximum size of any full binary tree that is a subtree of $T$.

**Exercise K.129.** For two polynomials $p(x) = a_0 + a_1 x + \cdots + a_n x^n$ and $q(x) = b_0 + b_1 x + \cdots + b_{n-1} x^n$, their product can be written as

$$p(x)q(x) = \sum_{i,j} a_i b_j x^{i+j}$$

where $i, j$ both range from 0 through $n$. Suppose we instead wanted to compute the following "skew-product",

$$\sum_{i<j} a_i b_j x^{i+j},$$

where again $i$ and $j$ both range from 0 through $n$, but restricted to $i < j$. Design and analyze a fast algorithm for computing the skew-product of two polynomials.

[53] It might simplify things to first apply the preceding exercise.

[54] As a second (appropriately vague) hint, how do you express "$a = b \vee c$" and "$a = b \wedge c$" in CNF?

**Exercise K.130.** Prove that every directed acyclic graph has at least one source, and at least one sink.

**Exercise K.131.** Let $G = (V, E)$ be an undirected graph. We say that a set of vertices is *almost independent* if each vertex $v \in S$ has at most one neighbor in $S$.[55] Consider the problem of computing the maximum cardinality of any almost independent set of vertices. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.132.** Recall the *1-in-3-SAT* problem from exercise K.186. Suppose instead that we had reduced from 1-in-3-SAT to subset sum. How might this have simplified our construction in Section 11.2?

**Exercise K.133.** Design and analyze a linear time algorithm for six-coloring a planar graph.

**Exercise K.134.** Recall that ants leave *trail pheromones* to help other ants follow its trail, particularly if the first ant identifies a good source of food.

Consider an $n \times n$ grid of cells identified by coordinates $\{(i, j) : 1 \leq i, j \leq n\}$. We say two cells are adjacent if they share a side. We are told an ant was placed in cell $(1, 1)$, and then walked around the grid, always from one cell to an adjacent cell, until stopping at some cell and then being taken off the grid. (The ant cannot move diagonally, hop over cells, and so forth. It may revisit the same cell.) The goal is to identify the final cell where the ant finished walking.

For every ordered pair of adjacent cells $(c, d)$ (where $c, d \in [n] \times [n]$), the pheromones allow us to deduce the number of times the ant moved from cell $c$ to cell $d$. We want to use this information to identify the final cell of the ant. We assume an *oracle* model where in a single query to the oracle, the user specifies an ordered pair of adjacent cells $(c, d)$, and the oracle returns the number of times the ant moved from cell $c$ to cell $d$. Design and analyze an algorithm to identify the final cell where the ant was lifted out of the grid. As always, the faster the better. We treat each call to the oracle as a constant time operation.

**Exercise K.135.** Recall that a sequence of numbers $x_1, \ldots, x_k$ is *(weakly) convex* if $x_{i+1} - x_i \geq x_i - x_{i-1}$ for $i = 2, \ldots, k-1$. The sequence $x_1, \ldots, x_k$ is *strictly convex* if $x_{i+1} - x_i > x_i - x_{i-1}$ for $i = 2, \ldots, k-1$.

Let $G = (V, E)$ be an unweighted directed graph with $m$ edges and $n$ vertices, where each vertex $v \in V$ is given a positive label $\ell(v) \in \mathbb{R}_{>0}$.

[55] Two vertices $u$ and $v$ are neighbors if they are connected by an edge.

1. (5 points) Consider the problem of computing the length of the longest path in $G$ where the vertex labels are weakly convex. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. (5 points) Now consider the problem of computing the length of the longest walk in $G$ where the vertex labels are strictly convex. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.136.** (Approximating subset sum.) Let $\epsilon \in (0, 1)$ be fixed. Here we treat $\epsilon$ as a fixed constant (like $\epsilon = .1$, for 10% error); in particular, running times of the form $O\left(n^{O(1/\epsilon)}\right)$ count as a polynomial.

A $(1 \pm \epsilon)$-approximation algorithm for subset sum is one that (correctly) either:

1. Returns a subset whose sum lies in the range $[(1 - \epsilon)T, (1 + \epsilon)T]$.

2. Declares that there is no subset that sums to (exactly) $T$.

Note that such an algorithm does not solve the (exact) subset sum problem. (As usual, we assume the input consists of natural numbers.)

1. Suppose every input number $x_i$ was "small", in the sense that $x_i \leq \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$-approximation algorithm for this setting.

2. Suppose every input number $x_i$ was "big", in the sense that $x_i > \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$-approximation algorithm for this setting.

3. Now give a polynomial time $(1 \pm \epsilon)$-approximation algorithm for subset sum in the general setting (with both big and small inputs).

**Exercise K.137.** Consider the following variation of subset sum which allows for an additive error of 1. We'll call it *subset-sum*$\pm 1$. The input consists of $n$ numbers $x_1, \ldots, x_n \in \mathbb{N}$ and a target $T \in \mathbb{N}$, like subset sum. The goal is to decide if there is a subset of $x_1, \ldots, x_n$ that sums to either $T - 1$, $T$, or $T + 1$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.138.** Here we assume the same model as exercise K.13 – an array $A[1..n]$ of incomparable elements[56] – and generalize the notion of a majority element. For $k > 1$, an element is a $(1/k)$-*heavy-hitter* if it has frequency at least $n/k$. (e.g., majority elements are $(1/2)$-heavy hitters.) Note that there are at most $k$ $(1/k)$-heavy-hitters in $A[1..n]$.

1. (2.5 pt.) Give a recursive spec for the $(1/k)$-heavy hitter problem.
2. (2.5 pt.) Give a divide-and-conquer algorithm for the $(1/k)$-heavy hitter problem implementing your recursive spec (the faster the better). (This algorithm should be similar to your majority element algorithm.)
3. (2.5 pt.) Analyze the running time of your algorithm.
4. (2.5 pt.) Prove your algorithm is correct by induction, where the recursive spec should act as your induction hypothesis.

Your solutions should generalize that of finding the majority elements. That is, plugging in $k = 2$ should recover the same results as for the majority problem.[57]

**Exercise K.139.** Recall that the *chromatic number* of a graph $G$, denoted $\chi(G)$, is the number number of colors needed to properly color the vertices of $G$. In *list coloring*, each vertex $v$ has a list $L(v)$ of permitted colors, and we seek a proper coloring where each vertex uses a color from its list. A graph $G$ is $k$-*choosable* if it has such a coloring whenever all lists have size at least $k$. The *list chromatic number* $\chi_{\text{L}}(G)$ is the minimum $k$ for which $G$ is $k$-choosable.

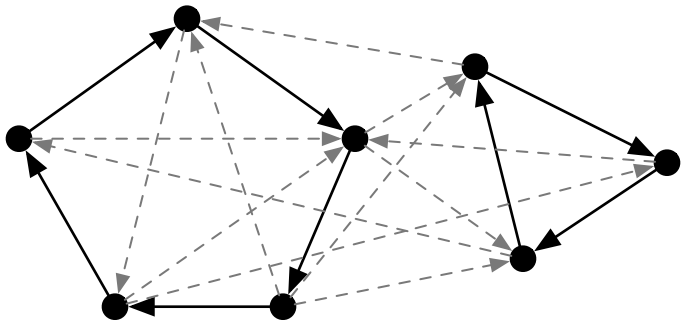
1. Prove that $\chi_{\text{L}}(G) \geq \chi(G)$ for any graph $G$.
2. Prove that every graph $G$ is $(\Delta + 1)$-choosable, where $\Delta$ is the maximum degree of $G$.
3. Prove that every tree is 2-choosable.
4. The *complete bipartite graph* $K_{n,n}$ has $2n$ vertices partitioned into two sides of $n$ vertices each, with edges between every pair of vertices on opposite sides. Prove that $\chi(K_{n,n}) = 2$.
5. Prove that $\chi_{\text{L}}(K_{n,n}) \geq \log_2 n$. (This is tricky.)

[56] In addition to disallowing sorting, no hash tables are allowed as well.
[57] Your algorithm should be faster than $O(n^2)$ (which is the trivial running time) for, e.g., $k = n^{1/3}$.

6. Prove the $\chi_{\text{L}}(K_{n,n}) \leq O(\log n)$. (This is also tricky; *randomization* might be helpful.)

**Exercise K.140.** Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function on $n$ variables. (Here no formula for $f$ is given, and we treat $f$ as a black box.) Prove that there is a CNF $\varphi(x_1, \ldots, x_n)$ of finite size such that $f = \varphi$.

**Exercise K.141.** A *cycle cover* of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover every vertex in $G$. You can also think of a cycle cover as a subset of edges $F \subseteq E$ such that each vertex $v$ has exactly one edge in $F$ entering $v$, and exactly one edge in $F$ leaving $v$. This implies that $F$ has exactly $n$ edges. Below is a directed graph where the solid arcs form a cycle cover, and the dashed arcs are the remaining edges in $G$.



Consider the problem of deciding if a directed graph $G$ contains a cycle cover. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.142.** This exercise is about how for many intents and purposes, we approximately have the extremely convenient identity, "$1 + x \approx e^x$".

1. Prove that for all $x \in \mathbb{R}$, $1 + x \leq e^x$.

   *Hint: At $x = 0$, both sides are equal. What are their respective rates of change moving away from 0?*

2. Prove that for all $x \leq 1$, $e^x \leq 1 + x + x^2$.

**Exercise K.143.** Most modern object-oriented programming languages use *garbage collection* to manage memory. In this model, we have a family of objects, each of which have pointers to other objects. We also have "external pointers" (on the stack) which point to some of the objects. At a given point in time, a computer program has direct access to objects via the external pointers on the stack, and then access additional objects by chasing/following pointers from one object to the next.

Clearly, when an object has no pointers to it (either external pointers, or pointers from other objects), then it is no longer accessible from the rest of the program, and it is safe to "garbage collect" the object and free up the underlying memory for the rest of the program to use. But more generally, any object that cannot be accessed by following pointers from object to object, starting from an external pointer, can be garbage collected since it cannot be accessed from the stack.

Garbage collection can have a cascading effect. When you garbage collect an object $x$, and delete its pointers, there may be more objects that are freed up. The goal is to identify these objects.

We can think of the object system as a directed graph $G = (V, E)$, where each vertex $v \in V$ is an object, and each arc $(u, v)$ models a pointer from object $u$ to object $v$. We model external pointers with a set $X \subseteq V$; each $x \in X$ indicates an external pointer to object $x$. "Chasing / following pointers" corresponds to a walk in this "object graph", and an object is garbage collected when it is no longer reachable from any $x \in X$.

For simplicity, you can assume that there are no duplicate pointers: there is at most one pointer $(u, v)$ from $u$ to $v$ for any pair $u, v \in V$; and at most one external pointer to each object. That is, both $X$ and $E$ are sets. (We can assume this without loss of generality, as one can always preprocess the input to enforce this.)

Suppose you have an object system described by the graph $G = (V, E)$ and the set of external pointers by $X$. We let $n = |V|$, $m = |E|$, and $\ell = |X|$. Given an external pointer $x \in X$, design and analyze an algorithm that identify the set of all objects that would be garbage collected upon removing $x$ from the system. (To be clear, the input consists of $G = (V, E)$, $X$, and $x$.)

**Exercise K.144.** Recall that the exclusive-or (XOR) of two boolean variables $x$ and $y$, denoted $x \otimes y$, is `true` iff exactly one of $x$ or $y$ is `true`.

In XOR-satisfiabilty, you are given a formula $f(x_1, \ldots, x_n)$ which is a conjuction of $m$ (binary) XOR's from the $n$ boolean variables and their negations; e.g.,

$$f(x_1, x_2, x_3, x_4) = (x_1 \otimes x_2) \wedge (x_2 \otimes \bar{x}_3) \wedge (x_3 \otimes x_4) \wedge (\bar{x}_1 \otimes \bar{x}_4);$$

and the goal is to compute a satisfying assignment for $f$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.145.** Each year, medical students apply to residency programs, and programs decide which applicants to accept. Both sides have preferences: students rank programs, and programs rank applicants. The National Resident Matching Program (NRMP) has used a stable matching algorithm since the 1950s to assign students to programs in a way that prevents "blocking pairs"—a student and program who would both prefer each other over their assigned match.

Formally, consider two disjoint sets $A$ and $B$ with $|A| = |B| = n$. Each element of $A$ has a strict preference ranking over all elements of $B$, and vice versa. A *perfect matching* is a bijection $M : A \to B$. A matching $M$ is *unstable* if there exist $a \in A$ and $b \in B$ such that $a$ prefers $b$ to $M(a)$ and $b$ prefers $a$ to $M^{-1}(b)$; such a pair $(a, b)$ is called a *blocking pair*. A matching with no blocking pair is *stable*.

The *Gale-Shapley algorithm* builds a perfect matching as follows. At a high level, each $a \in A$ applies to one $b \in B$ at a time, and each $b \in B$ tentatively accepts one of its applicants. It terminates when the tentatively accepted applications form a perfect matching.

In the first round, each $a \in A$ applies to its highest-ranked $b \in B$, and each $b \in B$ tentatively accepts its highest-ranked applicant, rejecting the rest. In each subsequent round, each rejected $a \in A$ applies to the best $b \in B$ it has not yet applied to. Each $b \in B$ tentatively accepts its most preferred among the new applicants and its current tentative match (if any), rejecting the rest.

1. Prove an upper bound on the number of rounds of the Gale-Shapley algorithm before terminating with a perfect matching.

2. Prove that the matching produced is stable.

3. Prove that each element of $A$ is matched to the best element of $B$ that they could be matched to in any stable matching.

4. Prove that each element of $B$ is matched to the *worst* element of $A$ they could be matched to in any stable matching.

The last two points imply that, if there is a unique stable matching, the algorithm finds it regardless of which side proposes.

**Exercise K.146.** Recall that given a sorted array $A[1...n]$ of comparable elements, we can find an element $x$ (i.e., identify an index $i$ such that $A[i] = x$, or declare that $x$ does not exist) in $O(\log n)$ time with binary search. Here we develop an extension that in some sense allows us to do binary search on infinite length arrays. (A situation which does arise naturally.)

For simplicity, we assume the elements in $A$ are distinct. For fixed $A$, and an element $x$, we say that $x$ has *rank $k$ in $A$* if $x$ would be the $k$th smallest element if it were an element in $A$. (If $x$ is already in $A$, then this is the unique index $k$ such that $A[k] = x$.)

The goal is design and analyze an algorithm with the same interface as binary search: given access to a sorted array $A[1..n]$, and an element $x$ either (a) returns the unique index $i$ such that $A[i] = x$, or (b) declares that $x$ is not in $A$. This time, the running time should take $O(\log k)$ time, where $k$ is the rank of $x$.

Note that $O(\log k) \leq O(\log n)$, but it may be significantly faster when $n$ is much larger than $k$, or even infinite.

**Exercise K.147.** Prove or disprove: For any two events $A, B$,

$$\mathbf{P}[A \wedge B] \leq \min\{\mathbf{P}[A], \mathbf{P}[B]\}.$$

**Exercise K.148.** Let $G = (V, E)$ be an undirected graph with distinct nonnegative edge weights $w : E \to \mathbb{R}$. For a spanning tree $T$, we say that the *bottleneck weight* of $T$ is the maximum weight edge in $T$, $\max_{e \in T} w(e)$.

1. Prove that the MST is also a minimum bottleneck weight spanning tree of $G$.
2. Design and analyze a $O(m + n)$-time algorithm for computing a minimum bottleneck weight spanning tree of $G$. (This is faster than any of our algorithms for MST.)[58]

**Exercise K.149.** For $p > 0$, the $L_p$-norm of a vector $x \in \mathbb{R}^k$ is defined by

$$\|x\|_p \stackrel{\text{def}}{=} \left( \sum_{i=1}^{k} |x_i|^p \right)^{1/p}.$$

[58] Here's step 1: compute the median edge weight in $O(m)$ time.

Two edge cases are $p = 0$ and $p = \infty$: here we define $\|x\|_0$ as the number of indices $i$ such that $x_i \neq 0$, and

$$\|x\|_\infty \stackrel{\text{def}}{=} \max_i |x_i|.$$

Let $G = (V, E)$ be an undirected graph with real-valued edge lengths $\ell : E \to \mathbb{R}$. We can take inspiration from $L_p$-norms to define the $L_p$-distance in graphs. For a walk $w$ with edges $e_1, \ldots, e_k$, and $p > 0$, we define the $L_p$-norm of $w$ as the $L_p$-norm of the vector of edge weights in $w$,

$$\|w\|_p \stackrel{\text{def}}{=} \left( \sum_{i=1}^{k} |\ell(e_i)|^p \right)^{1/p}.$$

Analogously we define the $L_0$ and $L_\infty$ norms of $w$ based on the definition for vectors. We define the $L_p$-distance (for $p > 0$, $p = 0$, and $p = \infty$) between two vertices $s$ and $t$ is the infimum of $\|w\|_p$ over all walks from $s$ to $t$.

Let $s, t \in V$ be fixed. Below, for different values of $p$, we consider the problem of computing the $L_p$-distance from $s$ to $t$ in $G$. For each of these problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

1. (2 pts.) The $L_0$-distance from $s$ to $t$.
2. (2 pts.) The $L_{1/2}$-distance from $s$ to $t$.
3. (2 pts.) The $L_1$-distance from $s$ to $t$.
4. (2 pts.) The $L_2$-distance from $s$ to $t$.
5. (2 pts.) The $L_\infty$-distance from $s$ to $t$.[59]

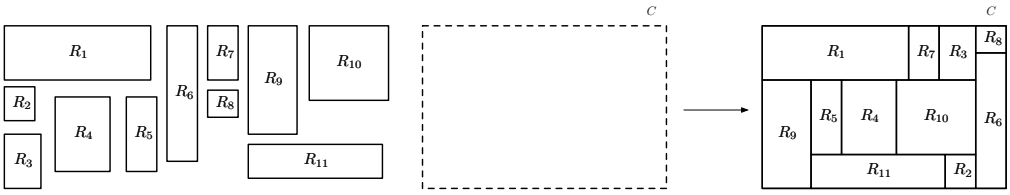
**Exercise K.150.** Let $I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]$ be a collection of $n$ intervals on the real line, and let $p_1, \ldots, p_m \in \mathbb{R}$ be a collection of $m$ points on the line with positive weights $w(p_i) > 0$. A point $p_i$ *hits* an interval $I_j$ if $p_i \in I_j$. A subset of points $p_{i_1}, \ldots, p_{i_k}$ is a *hitting set* for $I_1, \ldots, I_n$ if every interval $I_j$ is hit by some point $p_{i_h}$ in the subset. We assume that $p_1, \ldots, p_m$ itself is a hitting set. We also assume for simplicity that all the points $p_i$ are distinct. The goal is to compute the minimum weight of any hitting set of points. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

[59] For simplicity you may assume the edge weights are distinct.

**Exercise K.151.** Let $h : [n] \to [k]$ be *any* fixed function.

1. Prove that the number of collisions is
$$\geq \frac{n(n-k)}{2k}$$
2. Show that the above inequality is tight when $k$ divides $n$.

**Exercise K.152.** This problem is about rectangular *tiling*, where the input consists of a collection of (smaller) rectangular pieces, and a (larger) rectangular container. The high-level goal is to try to "tile" the rectangular container with the smaller rectangular pieces.




More formally, you have $n$ rectangular pieces $R_1, ..., R_n$ where each $R_i$ is specified by a height $H_i$ and a width $W_i$. (You can assume the input is given as two arrays $H[1..n]$ and $W[1..n]$). You are also given a rectangular container $C$ specified by a height $H_C$ and a width $W_C$. A "tiling" of $C$ by $R_1, \ldots, R_n$ is a placement of (some or all of) the rectangular pieces into the container so that there are no overlaps or gaps. (Like a jigsaw puzzle, or a tiled bathroom floor. To be clear, you can only use each piece once.) A "perfect tiling" of $C$ by $R_1, \ldots, R_n$ is a tiling where all the rectangular pieces are used. For simplicity we assume that you cannot rotate the pieces.

1. (5 points) Consider the problem of computing a tiling of $C$ by $R_1, \ldots, R_n$. (Not all the pieces have to be used.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. (5 points) Now consider the problem of computing a perfect tiling of $C$. (This time all the pieces have to be used.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.153.** Let $A[1..n] \in \mathbb{R}^n$ be an array of $n$ comparable elements. Let $k \in \mathbb{N}$ be an input parameter where $k \leq n$. (We are particularly interested in the case where $k$ is much smaller than $n$). Consider the problem where we want to obtained a sorted list of the $k$ smallest elements.

1. Design and analyze an algorithm (as fast as possible) that returns the $k$ smallest numbers in $A$ in sorted order.

2. Prove a lower bound (large as possible, up to constants) on the number of comparisons required by any correct algorithm.

Food for thought: How does your lower bound compare to your upper bound obtained in exercise K.153?

**Exercise K.154.** Kenjaku has initiated the *Culling Games*, a brutal competition designed to enforce his twisted vision of survival of the strongest. In the game, there are $m$ sorcerers and $n$ "cursed spirits", where $n$ is even. Kenjaku assigns a set of spirits to eliminate each sorcerer. A spirit may be assigned to multiple sorcerers.

Each sorcerer can only defend themselves against a limited number of spirits. (This number may vary by sorcerer.) The cunning Kenjaku strategically assigned exactly one more spirit to each sorcerer than they can handle.

As Satoru Gojo's successor, your goal is to save the overwhelmed sorcerers. To this end, you've partitioned the spirits into $n/2$ disjoint pairs, and you can destroy one spirit out of every pair. Your task is to choose one spirit from each pair to destroy so that each sorcerer can withstand the rest of the cursed spirits assigned to them.

Below are four different variations of the Culling Games varying in certain details. The input always consists of a list $S_i$ of spirits assigned to each sorcerer $i$, where the length of a list is always one more than the sorcerer can handle, and the partition of spirits into $n/2$ pairs (as a list of pairs). For each, determine whether there exists a polynomial-time algorithm, or whether such an algorithm would imply a polynomial-time algorithm for (Boolean) SAT.[60]

1. In the most general setting, each sorcerer can handle some given number spirits, and this number varies by sorcerer. Determine whether there exists a choice of one cursed spirit to destroy from each pair such that every sorcerer can defend themselves against the rest of the spirits.

[60] I (Kent) have no idea what's going on here, and I have never felt so old. I refer you to the teaching assistants for more background.

2. Suppose each sorcerer can fend off at most 1 spirit. (So Kenjaku assigns two spirits to each sorcerer.) Again, determine whether there exists a choice of one cursed spirit to destroy from each pair such that every sorcerer can defend themselves against the rest of the spirits.

3. Suppose again that each sorcerer can fend off at most 1 spirit, but you realized you cannot save all the sorcerers. (E.g., via the previous problem.) Consider the problem of choosing one spirit to destroy from each pair to save as many sorcerers as possible.

4. Suppose again that each sorcerer can handle at most 1 spirit (and therefore has two spirits assigned to attack them), but also needs to fight a spirit to justify their life choices. Determine whether there exists a choice of one spirit to destroy per pair such that every sorcerer has exactly one spirit remaining to fight.

**Exercise K.155.** A *partially ordered set* (or *poset*) is a set $P$ with a binary relation $\leq$ that is reflexive, antisymmetric, and transitive. A *chain* is a set of pairwise comparable elements; an *antichain* is a set of pairwise incomparable elements. The *width* of a poset is the maximum size of an antichain.

1. Prove that any partition of $P$ into chains requires at least $w$ chains, where $w$ is the width of $P$.

2. Suppose we want to partition $P$ into as few chains as possible. A natural greedy approach would take a maximal chain $C$, remove it, and repeat. Define a poset where this greedy approach uses more than $w$ chains.

3. Let $A$ be a maximum antichain. Let $U = \{x \in P : x \geq a \text{ for some } a \in A\}$ and $D = \{x \in P : x \leq a \text{ for some } a \in A\}$. Prove that $U \cup D = P$ and $U \cap D = A$.

4. Prove that any poset of width $w$ can be partitioned into exactly $w$ chains. (This is called *Dilworth's theorem*.)

**Exercise K.156.** An undirected graph $G = (V, E)$ is *bipartite* if one can partition $V$ into two sets $V_1, V_2$ such that every edge $e \in E$ has exactly one endpoint in $V_1$ and one endpoint in $V_2$. Consider the problem of deciding if an undirected graph $G$ is bipartite. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.157.** Tippecanoe County has decided to build a new hospital and call on you, seasoned and expert algorithmist, to help them figure out where to place it.

When planning hospitals, the high-level goal is to make sure that everyone in the town lives close to *some* hospital *on average*. We measure this by taking the average, over all homes in Tippecanoe Count, of the distance from the home to its nearest hospital. Tippecanoe County already has several hospitals[61]; your goal is to place the next hospital to improve this metric as much as possible.

Your input is a directed graph $G = (V, E)$ representing all the locations and roads in Tippecanoe county. Each edge $e$ has a label $\ell(e) > 0$ representing the length of that edge. Let $A \subseteq V$ denote the set of all current hospitals, $B \subseteq V$ the set of possible locations for new hospitals, and $C \subseteq V$ the set of all homes. Let $\alpha = |A|$, $\beta = |B|$, and $\gamma = |C|$. We assume that $A, B, C$ are disjoint and $\alpha \leq \beta \leq \gamma$. We also assume that $G$ is strongly connected.

1. (7 points) The first task is to evaluate the average distance (as described above) for the current set of hospitals. Consider the problem of computing the average distance from each home to its nearest hospital. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

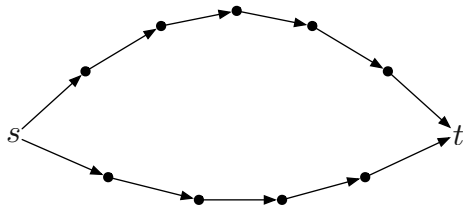
2. (3 points) Now consider the planning problem. The goal is to add one more hospital (from $Y$) to minimize the resulting average distance from each home to its nearest hospital. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

(For either problem, if you decide to design and analyze an algorithm, then your running times may depend on $\alpha = |A|$, $\beta = |B|$, and/or $\gamma = |C|$.)

**Exercise K.158.** Let $A[1..n] \in \mathbb{R}^n$ be an array of $n$ numbers. An *inversion* is a pair of numbers out of increasing order; more precisely, a pair of indices $i, j \in [n]$ such that $i < j$ and $A[i] > A[j]$. Design and analyze an algorithm (as fast as possible) for counting the number of inversions in $A$.

**Exercise K.159.** Let $G = (V, E)$ be a directed graph and $s, t \in V$. We say that two $(s, t)$-paths $p, q : s \rightsquigarrow t$ are *internally disjoint* if they are vertex disjoint except at their endpoints $s$ and $t$.

[61] In reality, Tippecanoe County has five hospitals. See https://www.in.gov/health/reports/QAMIS/hosdir/ctyfac78.htm




Consider the problem of computing the maximum number of edges of any two internally disjoint $(s,t)$-paths (for fixed $s$ and $t$). For example, the picture above gives two internally disjoint $(s,t)$-paths with a total of 11 edges. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.160.** For a graph $G$ and a set $S \subseteq V(G)$, let $\text{odd}(G-S)$ denote the number of connected components of $G-S$ with an odd number of vertices. The *deficiency* of $G$ is defined as

$$\text{defi}(G) \stackrel{\text{def}}{=} \max_{S \subseteq V(G)} (\text{odd}(G-S) - |S|).$$

1. Prove that $\text{defi}(G) \geq 0$ for any graph $G$.
2. Prove that every matching in $G$ leaves at least $\text{defi}(G)$ vertices unmatched.
3. Prove that if $G$ has a perfect matching, then $\text{odd}(G-S) \leq |S|$ for all $S \subseteq V(G)$.
4. Prove the converse: if $\text{odd}(G-S) \leq |S|$ for all $S \subseteq V(G)$ and $|V(G)|$ is even, then $G$ has a perfect matching. (This is *Tutte's theorem*, and is the trickiest part.)
5. Prove that the maximum matching in $G$ has size $\frac{1}{2}(n - \text{defi}(G))$. (This is called the *Tutte-Berge formula*.)

**Exercise K.161.** Recall the $O(n^3)$ dynamic programming algorithm for APSP. If the edge lengths were nonnegative, then we could have just run Dijkstra's algorithm from every vertex in $O(mn + n^2 \log n)$ time, which is never worse than $O(n^3)$, and significantly better when the graph is sparse. Here we will develop a classic algorithm due to Johnson [Joh77b] that builds on this observation. (Obviously you can find this algorithm in any book, but the point is that you have the background to discover it yourself.)

Let $G = (V, E)$ be a directed graph with real-valued edge weights $\ell : E \to \mathbb{R}$. For simplicity we assume there are no negative cycles in $G$. Let $d(s, t)$ denote the distance with respect to $\ell$. We define *vertex potentials* as a set of values $\varphi : V \to \mathbb{R}$ for each vertex. For a set of vertex potentials $\varphi$, consider the modified edge weights $\ell_\varphi : E \to \mathbb{R}$ defined by

$$\ell_\varphi(u, v) = \varphi(u) + \ell(u, v) - \varphi(v).$$

Let $d_\varphi$ denote distances with respect to $\ell_\varphi$.

1. Prove that for any walk $w : s \rightsquigarrow t$,

$$\bar{\ell}_\varphi(w) = \varphi(s) + \bar{\ell}w - \varphi(t).$$

Since all $(s, t)$-walks shift by the same quantity $\varphi(s) - \varphi(t)$, the shortest paths stay the same. We also have $d_\varphi(s, t) = \varphi(s) + d(s, t) - \varphi(t)$ for all $s, t$.

2. Consider the potentials defined by $\varphi(v) = \min_{s \in V} d(s, v)$. Prove that for these potentials, $\ell_\varphi$ is nonnegative.

3. Clearly the potentials above can be computed in $O(mn^2)$ time by running Bellman-Ford for every choice of source $s$. Design and analyze an algorithm computing $\varphi$ in $O(mn)$ time.

4. Finally, put everything together to obtain Johnson's $O(mn + n^2 \log n)$-time algorithm for APSP.

**Exercise K.162.** A Hadamard matrix is a family of square $\{-1, 1\}$-matrices $H_0, H_1, H_2$ defined recursively by:

$$H_0 = \begin{pmatrix} 1 \end{pmatrix} \in \mathbb{R}^{1 \times 1}$$
$$H_k = \begin{pmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{pmatrix} \in \mathbb{R}^{2^k \times 2^k}, \text{ for } k \geq 1.$$

1. Prove that for all $k$, $H_k H_k^\mathsf{T} = H_k^\mathsf{T} H_k = 2^k I$, where $I$ is the identity matrix.

2. Design and analyze a fast algorithm that, given $x \in \mathbb{R}^n$ for $n = 2^k$, computes the vector $H_k x \in \mathbb{R}^n$.

**Exercise K.163.** Subset Subs™, the world's first dynamically optimal deli, has the following delicious business model. Subset Subs makes only two submarine sandwiches each day:

- A ham sub $m$ inches long, and
- A vegetarian sub $n$ inches long,

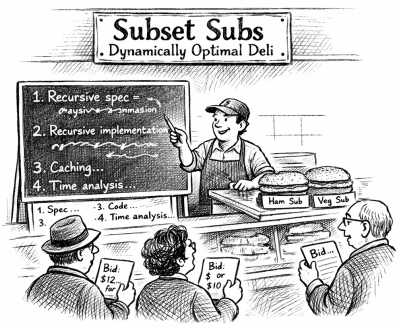
for integers $m$ and $n$, depending on market conditions. There are also plans to expand to a vegan option, but not hoagies.[62]

Subset Subs sells its sandwiches fractionally and dynamically as follows. Each day, $k$ customers submit bids on the sandwiches, each taking one of the following forms:

1. $x$ dollars for $y$ inches of the ham sandwich.
2. $x$ dollars for $y$ inches of the vegetarian sandwich.
3. A *disjunctive order*, specifying either $x_1$ dollars for $y_1$ inches of the ham sandwich, *or* $x_2$ dollars for $y_2$ inches of the vegetarian sandwich.

The disjunctive order means the customer would accept either of their listed options. The number of inches is always given as an integer.

There is usually too much demand to serve all of the bids. Each day, Subset Subs decides which of the bids to serve, with the objective of making the maximum amount of money. This means they choose a subset of ham orders, vegetarian orders, and disjunctive orders, choosing one of the two options for each disjunctive order they take, subject to having enough ham sandwich and enough vegetarian sandwich to serve all the orders they take. (We call this a "feasible" subset of orders.)



"Your sandwich will be ready in polynomial time."

In addition to the dollar amounts from the bids, Subset Subs also takes into account the ancillary costs when serving a piece of the sandwich, as a function of its length. This can reflect small things, like the wrapping paper, to the unusual logistics involved when someone orders an extremely large piece.

Ultimately, the input consists of the lengths $m$ and $n$; $k$ bids, each of one of the

[62] According the Guinness Book of World Records, the longest sandwich measured 735 m (2,411 ft 5 in) and was created by members of three teams in total. These teams were Groupe Notre Dame Hazmieh-Scouts de L'Independence (Lebanon), Municipality of Hazmieh (Lebanon)and Mini-B chain restaurants (Lebanon). The attempt took place in Hazmieh village, Beirut, Lebanon, on 22 May 2011.

The sandwich started at Notre Dame des Soeurs Antonines School and ended on Elie Street, in Hazmieh, Beirut, Lebanon. The width of the sandwich was 12.5cm and the overall estimated weight of the sandwich is 577.03kg.

three types above; and a lookup function $\texttt{cost} : \mathbb{Z}_{\geq 0} \to \mathbb{R}_{>0}$ that, given $x$, returns the ancillary cost of producing an $x$-inch long piece of sandwich, in $O(1)$ time. Design and analyze an algorithm to compute the maximum amount of profit that can be obtained by accepting a feasible set of orders. (As usual, the faster the better. Here we allow running times that are polynomial in $m$ and $n$, even though they are specified in $\log(m)$ and $\log(n)$ bits, respectively.)[63]

**Exercise K.164.** Fast forward 10 years and your LLM company GNAGI[64] is taking off. Your company's main product is an API where customers submit text generation tasks, but due to the Great GPU Panic you only have 2 machines and can't service all of the requests. So you've set up a bidding system, where each customer submits a job (with time specifications described below) along with a bid. The high-level task is to choose which of these jobs to schedule on your machines to maximize the total amount of money you make.

The input consists a sequence of $n$ jobs $J_1, \ldots, J_n$. Each job $i$ consists of a

- Time of arrival $T_i \geq 0$, a
- Duration $D_i > 0$, and a
- Bid $B_i > 0$.

For the sake of concreteness, we'll say that your input is given as three arrays $T[1..n]$, $D[1..n]$, and $B[1..n]$, listing the job arrival times, durations, and bids, respectively.

You get to choose which of the jobs to take on, and which machines to schedule them on. As mentioned above, you have 2 machines. Each machine can only work on one job at any given time. If you assign job $i$ on machine $M_j$ (where $j \in \{1, 2\}$), then the job occupies $M_j$ for $D_i$ units of time starting from time $T_i$, and you make a profit of $B_i$ dollars.

Consider the problem of computing the maximum profit (i.e., sum of bids) from scheduling jobs onto the 2 machines. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.165.** Recall the set cover problem for which we obtained a randomized $O(\log n)$-approximation. Here we consider a (maximum weight) set *packing* problem, defined as follows.

Let $[m]$ be a set of points, and let $S_1, \ldots, S_n \subseteq [m]$ be $n$ subsets of $[m]$. Let $b_1, \ldots, b_n > 0$ represent the profit of $S_1, \ldots, S_n$, respectively. We say that a collection

[63] You might find it helpful to first solve the simpler case of just one type of sandwich and no "disjunction orders".

[64] GNAGI stands for GNAGI Not AGI.

of sets $\mathcal{F} = \{S_{j_1}, \ldots, S_{j_k}\}$ is a *set packing* if they are all disjoint. The total profit of such a set packing is defined as the sum of profits $b_{j_1} + \cdots + b_{j_k}$ of the corresponding sets.

The goal is to compute a set packing of maximum profit, but the problem is NP-Hard. Here we consider the following (perhaps unusual) approximation criteria. Let OPT denote the maximum profit of any set packing. For $\alpha \geq 1$, we say that a collection of sets $S_{j_1} + \cdots + S_{j_k}$ is an $\alpha$-packing if each point is covered by at most $\alpha$ sets $S_{j_h}$. We say that a randomized collection of sets $\mathcal{F}$ is a *randomized approximate $\alpha$-packing* if

1. The expected total profit of $\mathcal{F}$ is at least OPT.
2. With high probability, $\mathcal{F}$ is an $\alpha$-packing.

Design and analyze a polynomial time algorithm that outputs a randomized approximate $\alpha$-packing for $\alpha$ as small as possible.[65]

**Exercise K.166.** Here we consider a special case of independent set that we will call *$k$-neighbors independent-set*, for a fixed parameter $k \in \mathbb{N}$. (e.g., $k = 5$.) The input is an undirected graph $G = (V, E)$ with $m$ edges and $n$ vertices, with the following property: every set $S \subseteq V$ has at most $k$ neighbors of $S$ outside of $S$.[66] The goal is to compute the maximum size independent set. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.167.** Let $G = (V, E)$ be a directed graph with $m$ edges, $n$ vertices, and nonnegative integer edge lengths $\ell : E \to \mathbb{Z}_{\geq 0}$. (Edges may have length 0.) Given two vertices $s$ and $t$, there may be many shortest paths (with respect to the edge lengths $\ell$) from $s$ to $t$. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

1. (5 points) Compute the minimum number of edges of any shortest $(s, t)$-path (with respect to $\ell$) from $s$ to $t$.
2. (5 points) Compute the maximum number of edges of any shortest $(s, t)$-path (with respect to $\ell$) from $s$ to $t$.

[65] Here we are interested in the approximation factor $\alpha$ – the smaller and closer to 1 the better – and not the exact polynomial running time.

[66] Here a vertex $v$ is a *neighbor* of $S$ if it is connected by an edge.

**Exercise K.168.** Consider a directed graph $G = (V, E)$ that is *transitively closed*. That is, a vertex $u$ can reach a vertex $v$ iff there is an edge from $u$ to $v$. Consider the problem of finding a Hamiltonian path in a transitively closed graph. That is, the input consists of a transitively closed graph, and the goal is to find a Hamiltonian path in the graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.169.** As you know well, this class is a lot: lots of students, lots of homework, and a huge staff. To divvy up the grading load, we want to divide the staff into two teams, Team Divide and Team Conquer, which will alternate homework grading week by week. Some staff are already committed to a team. The rest are flexible. The staff also has many pairwise friends between them. We prefer to keep friends together; friendly grading teams make for more friendly grading. (The teams do not have to be the same size.)

Formally, let $S$ denote the set of $n$ staff members. We have as input a list of $m$ pairwise friendships among the staff. We also have two disjoint subsets $D, C \subseteq S$ of staff members preassigned to Team Divide and Team Conquer, respectively. The goal is to assign the remaining staff members in $S \setminus (C \cup D)$ so as to minimize the number of friendships split across the teams. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.170.** Problems 1–5 describe a search problem on a nonempty array $A[1..n]$ of comparable elements.[67] For each of these problems, design a recursive algorithm that solves the search problem as fast as possible.[68] Briefly analyze the running time of each algorithm.[69]

1. Let $A[1..n]$ be in a *rotated* sorted order. That is, there exists an index $i \in [n]$, called the *rotation index*, such that the concatenation of $A[i..n]$ and $A[1..i-1]$ is sorted in increasing order. (One can think of $A[1..n]$ as being sorted initially, and then rotated cyclically to the right by $i - 1$ slots). The goal is to compute the unknown rotation index $i$. For simplicity, you may assume all the elements are distinct.

[67] Given two elements $x$ and $y$, you can query if $x < y$, $x > y$, or $x = y$.

[68] This includes a recursive spec, and pseudocode implementing the recursive spec.

[69] For the sake of brevity we do not ask for an explicit analysis of correctness. We will treat your recursive spec as the induction hypothesis of a proof by induction, and fill in the rest.

2. We say that $A[1..n]$ is a *mountain* if there exists an index $i \in [n]$, called the *peak*, such that $A[1..i]$ is sorted in increasing order and $A[i+1..n]$ is sorted in decreasing order. Given a mountain $A[1..n]$, find the peak. For simplicity, you may assume all elements are distinct.

3. We say an index $i \in [n]$ is a *local minimum* if either (a) $i = n = 1$, (b) $i = 1$ and $A[1] \leq A[2]$; (c) $i = n$ and $A[n] \leq A[n-1]$; or (d) $1 < i < n$, $A[i-1] \geq A[i]$ and $A[i] \leq A[i+1]$. The goal is to find a local minimum from an array $A[1..n]$.

4. Let $A[1..n]$ be an array of integers such that $A[1] = 0$ and $A[n]$ is odd. Call a consecutive pair of indices $(i, i+1)$ a *gap*. We say that a gap is *even* if the difference $A[i+1] - A[i]$ is even and *odd* if the difference is odd. Observe that when $A[1] = 0$ and $A[n]$ is odd, there must be an odd gap. Given an array of integers $A[1..n]$ such that $A[1] = 0$ and $A[n]$ is odd, find an odd gap.

5. Let $A[1..n]$ be an array of integers such that $A[1] = 1$ and $A[n] = n$. For $i \in \{1, \ldots, n-1\}$, we say that there is a *jump* at index $i$ if $A[i+1] \geq A[i] + 1$. Observe that since
$$\sum_{i=1}^{n-1} (A[i+1] - A[i]) = A[n] - A[1] \geq n - 1,$$
there must be a jump at least one index $i$. (If there was no jump, then the sum on the left would less than $n-1$, a contradiction.)

   Design and analyze an algorithm that, given an array $A[1..n]$ such that $A[1] = 1$, $A[n] = n$, and $n \geq 2$, finds an index $i$ for which there is a jump.

6. Prove that each of the problems above have a lower bound of $\Omega(\log n)$ queries in the comparison model.

7. (Extra credit / just for fun) Invent your own fun and interesting search problem on an array of comparable elements.

**Exercise K.171.** Prove or disprove: every walk in a directed acyclic graph has finite length.

**Exercise K.172.** Let $G = (V, E)$ be a directed graph with $m$ edges and $n$ vertices, where each vertex $v \in V$ is given an integer label $\ell(v) \in \mathbb{N}$. The goal is to find the length of the longest path[70] in $G$ where the labels of the vertices are (strictly) increasing.

[70] Recall that a path is a walk that does not repeat vertices.

1. Suppose $G$ is a DAG. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. Consider now the problem for general graphs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

3. Suppose instead we ask for the length of the longest path in $G$ where [71] is weakly increasing if $x_1 \leq x_2 \leq \cdots \leq x_k$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.173.** In *Minesweeper 2: On Graphs*, we have an undirected graph $G = (V, E)$ whose vertices are partitioned into *open* vertices $V_{\text{OPEN}}$ and *hidden* vertices $V_{\text{HIDDEN}}$. Hidden vertices may contain mines. Each open vertex tells you how many of its hidden neighbors are mines.

Each open vertex $v \in V_{\text{OPEN}}$ has a label $\ell(v) \in \mathbb{Z}_{\geq 0}$. A *mine placement* is a function $M : V_{\text{HIDDEN}} \to \{0, 1\}$. A mine placement is *consistent* if for every open vertex $v$,

$$\ell(v) = \sum_{\substack{w \in V_{\text{HIDDEN}} \\ \{v,w\} \in E}} M(w).$$

That is, each label equals the number of adjacent hidden vertices containing a mine.

Given $G = (V_{\text{OPEN}} \sqcup V_{\text{HIDDEN}}, E)$ and $\ell : V_{\text{OPEN}} \to \mathbb{Z}_{\geq 0}$, consider the problem of determining whether there is a consistent mine placement. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.174.** Let $G = (V, E)$ be a directed graph with vertex weights $w : V \to \mathbb{R}$. We say that the *vertex weight of a walk* going through vertices $v_1, v_2, \ldots, v_k$ (in order, with vertices possible repeating) is the sum weight $w(v_1) + w(v_2) + \cdots + w(v_k)$.

Let $k \in \mathbb{N}$ be a given parameter with $k \leq n$. Consider the problem of computing the minimum vertex weight of any walk with exactly $k$ vertices. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

[71] A sequence $G$ is a general graph and the labels of the vertices are weakly increasing.$x_1, \ldots, x_k$

**Exercise K.175.** Let $G = (V, E)$ be a connected graph with distinct edge weights and $k < n$. We define the $k$-MWF as the minimum weight forest among the forests with exactly $k$ edges. Consider the problem of computing the $k$-MWF.

1. Call an edge $e$ *$k$-great* if for every set of edges $F \subseteq E - e$ that (a) spans $e$ or (b) contains a forest of at least $k$ edges,

   $$w(e) < \max_{f \in F} w(f).$$

   Prove that $e$ is $k$-great iff every $k$-MWF contains $e$.

2. Design and analyze an algorithm for computing the $k$-MWF.

**Exercise K.176.** The *Ramsey number* $R(s, t)$ is the minimum $n$ such that every 2-coloring of the edges of $K_n$ (say, red and blue) contains either a red $K_s$ or a blue $K_t$. Our goal is to prove *Ramsey's theorem*: $R(k, k) \leq \binom{2k-2}{k-1}$ for all $k$.

1. Prove that $R(s, 2) = s$ for all $s \geq 2$.
2. Prove that $R(3, 3) = 6$.
3. Prove that $R(s, t) \leq R(s - 1, t) + R(s, t - 1)$ for $s, t \geq 3$.
4. Conclude that $R(s, t) \leq \binom{s+t-2}{s-1}$.
5. Prove Ramsey's theorem.
6. Prove that $R(k, k) > 2^{k/2}$. (This is trickier; *randomization* might be helpful.)

**Exercise K.177.** Let $G = (V, E)$ be a DAG, and $s, t \in V$. Consider the problem of computing the *number* of paths from $s$ to $t$ in $G$.[72] For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.178.** Suppose you repeatedly flip a coin that is heads with fixed probability $p \in (0, 1)$.

[72] Technically, there could be as many as $n!$ paths in which case each arithmetic operation would take $O(\log(n!)) = O(n \log n)$ time. For this exercise, let us assume for simplicity that arithmetic operations take $O(1)$ time anyway.

1. What is the expected number of coin flips until you obtain one heads?[73] Prove your answer.
2. What is the expected number of coin flips until you obtain two heads? Prove your answer.
3. For general $k \in \mathbb{N}$, what is the expected number of coin tosses until you obtain $k$ heads? Prove your answer.

**Exercise K.179.** Here we consider a vast generalization of edit distance, as follows. Suppose the cost of an edit sequence is given by a nonnegative real-valued function $f(a, b, c) \geq 0$, where $a$ is the number of insertions, $b$ is the number of deletions, and $c$ is the number of substitutions. We assume that the function is monotonically increasing in each of its arguments. That is, increasing $a$, $b$, or $c$ increases the value of $f(a, b, c)$.

Assume that $f$ is provided as a subroutine that takes $O(1)$ time to evaluate. Consider the problem the minimum cost edit sequence w/r/t the cost function $f$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.180.** Recall that a linked-list based stack supports the `push(`$x$`)` and `pop()` operations in $O(1)$ worst-case time. Consider the following operation, called `multipop`, that takes as input $k \in \mathbb{N}$, and removes and returns the top $k$ items on the stack.

`multipop(stack` $S$`,` $k$`)`

1. $L \leftarrow$ empty linked list
2. For $i = 1, \ldots, k$ as long as $S$ is not empty:
   A. Let $x = S$`.pop()`, and attach $x$ to the top of the list.
3. Reverse $L$ and return it.

Prove that after incorporating `multipop`, all three operations `push`, `pop`, and `multipop` each take $O(1)$ amortized time.

[73] If the first toss is heads, that counts as one coin flip. If the first toss is tails and the second toss is heads, that counts as two coin tosses. Etc. It may be helpful to first think about a fair coin, where $p = 1/2$.

**Exercise K.181. Can't we all just get along?** You want to throw a big party to celebrate finishing midterm 2, but not all of your friends get along. So you resolve to invite as many friends as possible while making sure there are no problems within the party.

Formally, we assume that you have $n$ friends indexed by $1, \dots, n$. Your input consists of a list $L$ of length $m$ of all pairs of friends who don't get along. The goal is to compute the largest cardinality (of any) subset of friends, $S \subseteq [n]$, that does not contain any pair of friends in the list $L$.

1. (5 points) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. (5 points) Now suppose there are no "cycles" among the pairs of enemies: that is, there is no sequence of friends $f_0, \dots, f_{k-1}$, where $k \geq 3$, such that $f_i$ does not get along with $f_{i+1 \mod k}$ for all $i$. For this setting, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.182.** Let $G = (V, E)$ be a directed graph with no parallel edges. We say that a walk $w$ in $G$ has *distinct edges* if no edge appears twice in $w$. Consider the problem of computing the length of the longest walk in $G$ with distinct edges. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.183.** Let $p(x, y) = \sum_{i=0}^{m} \sum_{j=0}^{n} a_{ij} x^i y^j$ and $q(x, y) = \sum_{i=0}^{m} \sum_{j=0}^{n} b_{ij} x^i y^j$ be two multivariate polynomials each with maximum degree $m$ in $x$ and maximum degree $n$ in $y$. Consider the product $r(x, y) = p(x, y)q(x, y)$, which has maximum degree $2m$ in $x$ and maximum degree $2n$ in $y$. It is easy to see that $r$ can be computed in $O(m^2 n^2)$ time with nested loops. Design and analyze a faster algorithm computing the product $r$.

**Exercise K.184.** John McCarthy[74] invented Lisp in 1960. Lisp has an extraordinarily simple syntax: a Lisp program is a nested composition of *S-expressions*. An S-expression is either

1. an *atom*: a single token representing a name or value (such as `x`, `42`, or `nil`), or

[74] McCarthy's 1971 Turing Award lecture, "Generality in Artificial Intelligence," is available at https://dl.acm.org/doi/epdf/10.1145/33447.33448.

2. a *list*: written in prefix notation as $(f\ e_1\ e_2\ e_3\ \cdots\ e_k)$ where $f$ is a token (the *operator*) and $e_1, \ldots, e_k$ are S-expressions (the *arguments*). In this problem we restrict to $k \in \{0, 1, 2\}$.[75]

This uniform representation, in which code and data are the same kind of object, makes Lisp *homoiconic*, and is the source of much of its expressive power.[76]

The parentheses make the structure of a Lisp program completely unambiguous. But suppose a Lisp program has been "flattened" and lost all of its parentheses, leaving only the flat sequence of tokens read left to right. For example, the S-expression

```
(cons (quote x) (quote y))
```

flattens to the sequence

```
cons quote x quote y
```

Can we recover a valid S-expression from such a sequence?

We model this as follows. Let $C[1..n]$ be an array of $n$ tokens. We say that $C[1..n]$ is *feasible* if there exists a single S-expression $e$ where the flattened sequence of $e$ equals $C$. We say that $C$ is *uniquely feasible* if the choice of $e$ is unique.

Suppose you have access to a constant time function `type` that returns, for each token $C[i]$, the set of roles it can play:

$$\texttt{type}(C[i]) \subseteq \{\texttt{atom}, 0, 1, 2\}.$$

Here `atom` means the token can serve as an atom, and $k \in \{0, 1, 2\}$ means it can serve as an operator taking $k$ arguments. A single token may have multiple roles; for example, a token like "`-`" might have one argument (`(- x)` negates $x$) or two (`(- x y)` subtracts $y$ from $x$). A 0-argument operator represents a function that takes no arguments.

1. (9 points) Design and analyze an algorithm (the faster the better) that, given $C[1..n]$, decides whether $C$ is feasible.

2. (1 point) Briefly describe how to modify your algorithm to decide whether $C$ is *uniquely* feasible.

[75] For simplicity, our operators are restricted to single tokens, and not nested S-expressions.

[76] See "The Roots of Lisp", by Paul Graham, available at https://paulgraham.com/rootsoflisp.html

**Exercise K.185.** After your glorious app PikPok hit number 1 in the app store, you're preparing for version 2. Obviously, it needs to be great.

You've gathered a list of $k$ features $F_1, ..., F_k$ that you could potentially add to version 2. However, there are complicated dependencies and requirements among them so you don't necessarily want to add all of them. There are 3 types of specifications defined over pairs of features $F_i$ and $F_j$:

1. Requirements: Your app must include either $F_i$ or $F_j$.
2. Conflicts: You cannot include both $F_i$ and $F_j$.
3. Dependencies: If you include $F_i$, then you must include $F_j$.

Collectively, we call requirements, conflicts, and dependencies the *feature specifications*. The feature specifications are given in list form. The high-level task is to decide which of the features to implement, based on the given feature specifications. We have two versions of the problem:

1. In the idealistic feature selection problem, the task is to decide if there is a subset of features that satisfies all the feature specifications.
2. In the realistic feature selection problem, the task is to choose a subset of features that satisfies the maximum number of feature specifications.

For each of the problems above, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.186.** In the *1-in-3-SAT* problem, one is given a 3-CNF $f(x_1, \ldots, x_n)$. The goal is to find a satisfying assignment such that every disjunctive clause (with three variables) is satisfied by *exactly* one of the variables.

Either (a) show that a polynomial time algorithm for 1-in-3-SAT implies a polynomial time algorithm for SAT, or (b) design and analyze a polynomial time algorithm for 1-in-3-SAT.

**Exercise K.187.** You have a sequence of $n$ switches $S_1, \ldots, S_n$ that jointly control $m$ light bulbs $L_1, \ldots, L_m$. Each switch can be "up" or "down", and this controls whether the light bulbs are on or off.

Each light bulb $L_i$, is associated with two sets of switches $A_i, B_i \subseteq [n]$. The switches in $A_i$ turn on the light bulb when they are "up" and the switches in $B_i$ turn on the light bulb then they are "down".

More precisely, for each $j \in A_i$, having switch $S_j$ "up" automatically turns on the light bulb. (It only takes one of these switches to be "up" to turn on the light bulb.) For each $j \in B_j$, turning the switch "down" automatically turns on the light bulb. (Again, it only takes one of these switches to be "down" to turn on the light bulb.)

Thus, for a light bulb $L_i$, the light bulb $L_i$ lights up if and only if either (a) some switch in $A_i$ is flipped up or (b) some switch in $B_i$ is flipped down. $A_i$ and $B_i$ are generic subsets of switches, not necessarily disjoint, and their union does not necessarily include all the switches. We do assume, however, that $|A_i| + |B_i| \geq 2$ for all $i$. We assume that the sets $A_i$ and $B_i$ are given explicitly for each $i$ (for simplicity; otherwise they can be obtained by inspection).

Your algorithm can flip switches "up" and "down". For the sake of running times, assume that flipping a single switch takes $O(1)$ time, and inspecting whether a single light bulb is on or off takes $O(1)$ time. The light bulbs turn on and off instantly when you flip a switch.

For each of the following decision problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

1. Decide if there exists a way to flip the switches to turn on all the light bulbs.
2. Decide if there exists a way to flip the switches to turn on at least three-fourths of the light bulbs.

**Exercise K.188.** Of the multiple algorithms we have seen for MST, recall the parallel approach (Borůvka's algorithm) and the search approach (Prim's algorithm) with Fibonacci heaps. There is also a hybrid approach running in $O((m+n)\log\log n)$ time where one runs the parallel algorithm for some number of iterations (say, $k$ iterations for a parameter $k$ TBD) and contracts the selected edges, and then runs the search algorithm thereafter. (Such an algorithm is correct because ultimately it only takes good edges.) Describe and analyze this algorithm.[77]

**Exercise K.189.** Let $G = (V, E)$ be a directed graph. Design and analyze an algorithm for each of the following problems.

1. Decide if there is a vertex $x$ such that $x$ can reach all other vertices.
2. Decide if there is a vertex $x$ such that $x$ can be reached by all other vertices.

[77] It may be helpful to obtain a running time generically as a function of $k$, and then optimize $k$.

3. Decide if there is a vertex $x$ such that every vertex can either reach $x$ or be reached from $x$.

**Exercise K.190.** Let $G = (V, E)$ be a directed graph with $m$ edges and $n$ vertices. Recall that for $s, t \in V$, $s$ can *reach* $t$ if there is a path from $s$ to $t$. There are many efficient algorithms to figure out if $s$ can reach $t$, that run in $O(m + n)$ time but also use $O(n)$ space.

Suppose we want to develop an algorithm that uses polylogarithmic space (but may take more than polynomial time). Consider the following recursive specification:

> `reach(`$s$`,`$t$`,`$k$`)`: Given two vertices $s, t \in V$ and $k \in \mathbb{N}$, returns true if $s$ can reach $t$ in at most $k$ edges.

Design and analyze a recursive implementation of `reach` that runs in subexponential time and uses polylogarithmic space.

**Exercise K.191.** Recall that a *palindrome* is a string spelled the same forward or backwards, such as "racecar". Likewise we say that a bit string $x \in \{0,1\}^n$ if $x_i = x_{n+1-i}$ for all indices $i = 1, \ldots, n$. Describe a SAT formula in CNF of size $O(n)$ for a Boolean formula $f : \{0,1\}^n \to \{0,1\}$ that accepts palindromes (i.e., $f(x) = 1$ iff $x$ is a palindrome).

**Exercise K.192.** Let $G = (V, E)$ be a directed graph and $s, t \in V$, where we assume there is no edge from $s$ to $t$. A collection of $(s,t)$-paths $p_1, \ldots, p_k$ are said to be *vertex-disjoint* if the interior vertices (excluding $s$ and $t$) of the paths are disjoint. A set of vertices $C \subseteq V \setminus \{s, t\}$ is a *vertex* $(s,t)$*-cut* if removing $C$ (and all incident edges) from $G$ disconnects $s$ from $t$. Consider the problems of (a) computing the maximum cardinality collection of vertex disjoint $(s,t)$-paths, and (b) the minimum vertex $(s,t)$-cut. For both of these problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.193.** Let $P$ be a set of $n$ distinct points in the plane. Consider the problem of computing the minimum distance $\|p - q\|$ between any two distinct points $p, q \in P$. (If $|P| \leq 1$, the minimum distance is defined as $+\infty$.) For simplicity we assume the points are in general position, so that all pairwise distances are unique. You may have learned a deterministic divide-and-conquer algorithm that runs in $O(n \log n)$ time. Here we will analyze the running time of a different randomized algorithm. To help build intuition, first solve the following problem.

1. (3 points) Let $a_1, \ldots, a_n \in \mathbb{R}$ be $n$ distinct numbers presented in a uniformly random order. Imagine tracking the minimum as you examine the numbers one by one in (the randomized) order. What is the expected number of times the minimum changes?

We assume black box access to a data structure for the following "threshold" version of the minimum distance problem. Fix a threshold $\rho > 0$. The data structure takes $O(1)$ time to initialize, and starts with an empty point set. It supports the following operation:

> `insert(`$x$`)`: Insert a point $x \in \mathbb{R}^2$ into the underlying point set. If the minimum distance in the underlying point set is (strictly) less than $\rho$, then return `true`; otherwise return `false`. This operation takes $O(1)$ expected time.[78]

This subroutine, combined with part 1, suggests the following algorithm. Here we assume that one can do math operations (add, subtract, square, etc.) in $O(1)$ time, and that we can sample a random permutation of $n$ items in $O(n)$ time.

1. Let $p_1, p_2, \ldots, p_n$ permute $P$ uniformly at random. *// $O(n)$ time.*
2. Let $r_1 = +\infty$ and $r_2 = \|p_1 - p_2\|$.

*/* We maintain the invariant that $r_i$ is the minimum distance over $p_1, \ldots, p_i$. */*

3. Start a new instance of the threshold data structure with threshold $r_2$. Insert $p_1$ and $p_2$.
4. For $i = 3, \ldots, n$:
   A. Insert $p_i$ into the threshold data structure. If the data structure returns true:
      1. Let $r_i = \min\{\|p_i - p_j\| : j < i\}$.
      2. Replace the threshold data structure with a new one with radius $r_i$. Insert $p_1, \ldots, p_i$ into the data structure.
   B. Otherwise set $r_i = r_{i-1}$.
5. Return $r_n$.

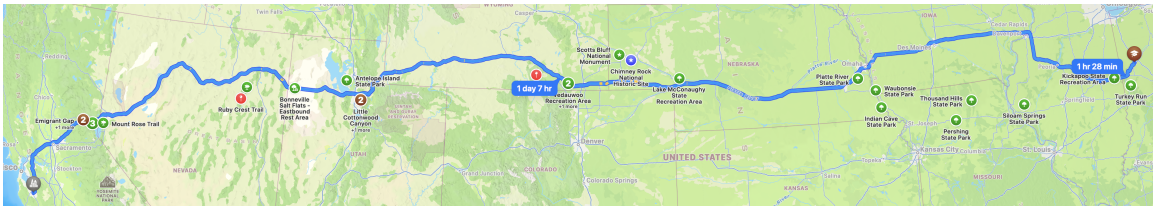
[78] We can implement this data structure by building a "hash table over grid cells". Initially we have an empty dictionary for an empty point set. Given a point $x = (x_1, x_2)$, compute the key `key(`$x$`)` $= (\lfloor x_1/(\rho/\sqrt{2}) \rfloor, \lfloor x_2/(\rho/\sqrt{2}) \rfloor$ in $O(1)$ time. (Rounding is not a standard operation in all models of computation, but here we assume it takes $O(1)$ time.) If this key is already occupied by a point $y \in P$, then $\|x - y\| \le \rho$, and we return `true`. Otherwise consider the 8 "neighboring" keys/cells where we add or subtract at most one or zero to each of the key's coordinates. Assuming the minimum distance was $\ge \rho$ before inserting $x$, each of these 8 cells can have at most 1 point. If the distance between $x$ and any of the $O(1)$ points in neighboring cells is $< \rho$ from $x$, return `true`.

Now analyze this algorithm via the following steps.

2. (3 points) For each index $i > 2$, analyze the exact probability that $r_i < r_{i-1}$.

3. (4 points) Finally, bound the expected running time of the algorithm.

**Exercise K.194.** Imagine you are planning a long road trip to San Jose, CA and you've already fixed your route (through Illinois, Iowa, Nebraska, Wyoming, Utah, Nevada, and then California via Tahoe).




You don't mind the drive, but you hate having to stop to get gas. What a pain! And some stations take longer than others. The car can drive at most 500 miles before needing to stop to get gas. The high-level goal is to plan your gas stops to minimize the amount of time spent pumping gas, while making sure you don't run out of gas between them.

We let $n$ denote the total number of gas stations, and number the gas stations $1, \ldots, n$ in order along the route. The input consists of two arrays $D[1..n]$ and $T[1..n]$, and an additional value $Z > 0$. For each $i$, $D[i] \in \mathbb{R}_{\geq 0}$ is a nonnegative real number representing the distance along the route at which the gas station appears. (Again, the gas stations are ordered to be nondecreasing in $D[i]$.) $T[i]$ is a positive real number representing the amount of time it takes to pump gas at station $i$. $Z$ represents the distance to the destination. You start with a full tank of gas.

Design and analyze an algorithm to compute the minimum amount of time one has to spend at gas stations in going from location 0 to location $Z$, while ensuring that you never drive more than 500 miles without visiting a gas station. (You should return $+\infty$ if it is impossible to do so.)

**Exercise K.195.** For CIPs, we could assume without loss of generality that $A_{ij} \leq b_j$ for all $i, j$. (In fact this assumption was critical for applying the Chernoff bound.) Suppose now that we had $\lambda A_{ij} \leq b_j$ for all $i, j$ for a parameter $1 \leq \lambda \leq \log(n)$. Design and analyzing a $O(\log(m)/\lambda)$-approximation algorithm for this setting.

**Exercise K.196.** Suppose we are given a two-dimensional array $A[1..m][1..n]$ of non-negative real values such that each row and column sum is an integer. We want to round $A$ to an integer matrix, replacing each entry $x \in A$ with either $\lceil x \rceil$ or $\lfloor x \rfloor$, while maintaining the sum in any row or column of $A$. For example,

$$\begin{pmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{pmatrix} \text{ rounds to } \begin{pmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{pmatrix}.$$

Consider the problem of computing such a rounding, or declaring that none exists. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.197.** Let $X$ and $Y$ be two sets of integers. We define the *unique sums* of $X$ and $Y$ as the set of integers of the form

$$z = x + y$$

where $x \in X$, $y \in Y$, and the choice of $(x, y) \in X \times Y$ is unique.[79] You may assume that all the integers in $X$ are distinct (amongst themselves), and that all the integers in $Y$ are distinct (amongst themselves).[80]

1. (2 pt.) Suppose $X$ and $Y$ each have $n$ integers. Design and analyze a $O(n^2 \log(n))$ time algorithm to compute the unique sums of $X$ and $Y$.

2. (6 pt.) Suppose all the integers in $X$ are $Y$ are between 0 and $M$ for a fixed value $M \in \mathbb{N}$. Design and analyze a $O(M \log M)$ time algorithm to compute the unique sums of $X$ and $Y$.[81]

3. (2 pt.) Suppose that now we had $k$ sets $X_1, \ldots, X_k$, each consisting of integers between 1 and $M$. (You may again assume no duplicates within each $X_i$.) A unique sum of $X_1, \ldots, X_k$ is defined as an integer of the form

   $$x_1 + \cdots + x_k,$$

   where $x_1 \in X_1$, $x_2 \in X_2$, $\ldots$, $x_k \in X_k$, and the choice of $(x_1, \ldots, x_k) \in X_1 \times \cdots \times X_k$ is unique. Design and analyze an algorithm that computes the unique sums of $X_1, \ldots, X_k$ in $O(kM \log(Mk) \log(k))$ time. You may assume for simplicity that $k$ is a power of 2.[82]

[79] That is, for fixed $z$, there is only one choice of $x \in X$ and one choice of $y \in Y$ such that $z = x + y$.

[80] For example, for $X = \{0, 1\}$ and $Y = \{0, 1\}$, the unique sums are $\{0, 2\}$. 1 can be obtained by $0 + 1$ or $1 + 0$ so it is not a unique sum.

[81] It might be helpful to work through very simple examples such as $X = Y = \{1\}$ and $X = Y = \{0, 1\}$.

[82] It might help to work through $k = 4$ to build your intuition.

*Hint: For part 2, try "changing the input" to an algorithm from Section 4.3 (that is not the FFT, at least directly).*

**Exercise K.198.** Let $A[1..n]$ be an array where each entry is colored `red` or `blue`, such that $A[1]$ and $A[n]$ have different colors. (This also implies that $n > 1$.) We say that a pair of consecutive indices $(i, i+1)$ is *colorful* if $A[i]$ and $A[i+1]$ have different colors. Design and analyze an algorithm (the faster the better) that computes a colorful consecutive pair of indices $(i, i+1)$ in $A$, or `nil` if there is no colorful pair of indices.

**Exercise K.199.** Consider an instance of (weighted) set cover defined by sets $S_1, \ldots, S_n \subseteq [m]$ and costs $c_i > 0$ for each set $S_i$. The goal is to compute the minimum cost collection of sets covering $[m]$. We saw that solving the LP and then randomly rounding gives a $O(\log m)$ approximation. Here we consider a special case where all the sets are small and obtain a better approximation factor by a standard extension of randomized rounding called *alterations*.

Let $\Delta \in \mathbb{N}$ be such that $|S_j| \leq \Delta$ for all $j$. Consider the algorithm `round-and-fix` for which some pseudocode is given below. `round-and-fix` is similar to randomized rounding and has two stages. The first stage solves the LP and then *rounds* the solution scaled up by some factor $\alpha \geq 1$. It is possible that some of the elements $i \in [m]$ may not be covered. In the second stage, we *fix* each uncovered element by (deterministically) taking the cheapest set that covers it.

`round-and-fix(sets` $S_1, \ldots, S_n \subseteq [m]$`, costs` $c \in \mathbb{R}^n_{>0}$`,` $\alpha \geq 1$`)`

1. let $x \in [0,1]^n$ solve the set cover LP
2. let $F \subseteq \{S_1, \ldots, S_n\}$ sample each set $S_i$ independently with probability $\min\{1, \alpha x_i\}$
3. for each $i \in [m]$
   A. if $i$ is not covered by $F$
      1. add the cheapest set covering $i$ to $F$
4. return $F$

Show that for an appropriate choice of $\alpha$, this algorithm returns a $O(\log \Delta)$ approximation to the set cover instance (in expectation). (It is possible to get $\log \Delta + \log\log \Delta + O(1)$ with care.)

**Exercise K.200.** Recall the `augmenting-paths` algorithm for flow which augments along arbitrary augmenting paths. Design an $(s,t)$-flow problem with integer capacities where this algorithm may take exponentially many iterations to terminate.

**Exercise K.201.** Let $G = (V, E)$ be an undirected graph, and let $a, b, c \in V$ be three distinct vertices. We define an $(a, b, c)$-*path* as a path from $a$ to $c$ that goes through $b$. Consider the problem of deciding if there exists an $(a, b, c)$-path. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.202.** Let $A[1..m]$ and $B[1..n]$ be two arrays. A common subsequence of $A$ and $B$ is a sequence that is a subsequence of both. Design and analyze an algorithm for each of the following problems, addressing Items 1–5 from Section 5.2.[17]

1. Compute the length of the longest common subsequence of $A$ and $B$.
2. Compute the length of the shortest common supersequence of $A$ and $B$.
3. Compute the length of the longest common subsequence of $A$ and $B$ that is a palindrome.
4. Suppose $A$ and $B$ consist of integers. Compute the length of the longest common increasing subsequence of $A$ and $B$.
5. Compute the length of the shortest common palindrome supersequence of $A$ and $B$.
6. Suppose $A$ and $B$ consist of integers. Compute the length of the longest common convex subsequence of $A$ and $B$. (Cf. problem 2)

**Exercise K.203.** Recall that different cost models are designed to reflect assumptions on how edits are naturally generated. For example, a reasonable hypothesis is that $k$ consecutive deletions, or $k$ consecutive insertions, are more likely than $k$ independent single-character insertions or deletions. For example, in writing these notes, the author frequently finds himself deleting entire sentences[83] at a time. We can factor this propensity into our edit distance model by a "insert-or-delete-at-bulk" cost. Let $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a fixed nonnegative function. We introduce the operations of "bulk insertion" and "bulk deletion", where one deletes $k$ characters or inserts a string of

[83] or paragraphs, or more... sigh...

$k$ characters, at a cost of $f(k)$, for any $k \in \mathbb{N}$. *We assume that $f$ is monotonically increasing in $k$; that is, $f(k) < f(k+1)$ for all $k$.* We can then define the *bulk edit distance* of two strings $x$ and $y$ to be the minimum cost of any sequence of edit, bulk insertion, or bulk deletion operations.

Assume that $f$ is provided as a subroutine that takes $O(1)$ time to evaluate. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.204.** An *edge coloring* of a graph is an assignment of colors to edges such that no two edges sharing a vertex have the same color. The *edge chromatic number* $\xi(G)$ is the minimum number of colors needed in an edge coloring of $G$.

1. Prove that $\xi(G) \geq \Delta$ for any graph $G$ with maximum degree $G$.
2. Let $G$ be a $\Delta$-regular bipartite graph. (Every vertex has degree $\Delta$). Prove that $\xi(G) = \Delta$.[84]
3. Now let $G$ be a bipartite graph with maximum degree $\Delta$. Prove that $\xi(G) = \Delta$. (This is called *König's theorem.*)

**Exercise K.205.** The following problem (and more elaborate extensions) appear in reinforcement learning. Let $G = (V, E)$ be a directed graph and let the edges be annotated by positive edge weights $r : E \to \mathbb{R}_{>0}$. We think of the vertices as states of some device under our control, and the edge weights $r(e)$ as rewards obtained by traversing the edge $e$ as follows. Let $s \in V$ be a fixed starting vertex / state.

1. Given an integer $k \leq n$, the goal is to compute a walk of length at most $k$ maximizing the sum of rewards along that walk. (If you repeat an edge, you get the same reward each time you repeat the edge.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.
2. Here we also incorporate a *discount rate*. Let $\alpha \in (0, 1)$ be given. Given a walk with edges $e_1, \ldots, e_k$, the *discounted total reward* of the walk is given by

$$r(e_1) + \alpha r(e_2) + \cdots + \alpha^{k-1} r(e_k).$$

[84] Hint: you might want to do exercise K.24 first.

The idea is that if we think of each edge traversal as also taking a unit of time, then the rewards attained far off in the future are perceived to be worth less than the rewards attained now and in the short term.

Given an integer $k \leq n$, the goal is to compute a walk of length at most $k$ maximizing the discounted total reward of the walk. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.206.** Recall that a sequence of numbers $y_1, y_2, \ldots, y_k$ is *increasing* if $y_i < y_{i+1}$ for each $i$. Recall from MCQ that a common subsequence of two arrays $A$ and $B$ is another sequence that is a subsequence[85] of both $A$ and $B$. Consider the problem of computing the length of the *longest common increasing subsequence* of two arrays of numbers, $A[1..m]$ and $B[1..n]$. For example,

$$(1, 4, 5, 6, 7, 9)$$

is the longest common increasing subsequence of the sequences

$$(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3) \text{ and } (1, 4, 1, 4, 2, 1, 3, 5, 6, 2, 3, 7, 3, 0, 9, 5).$$

For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.207.** The Economist recently reported[86] that the number of toll roads in America is increasing rapidly. It used to be hard enough to find the shortest path. Now you have to do it within a budget.

Right now, drivers pay to access 6,300 miles of Americas 160,000 miles (or so) of highway, but that number is going up. Here in Indiana, in June, the state passed a law giving the state authority to raise tolls on all its existing interstate highways. The Economist says that this trend is broadly due to a parallel trend of decreasing taxes on gasoline, which used to pay a larger share of the cost for maintaining roads and highways.

[85] As always, a subsequence is anything obtained from a sequence by extracting a subset of elements, but keeping them in the same order; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings `C`, `DAMN`, `YAIOAI`, and `DYNAMICPROGRAMMING` are all subsequences of the string `DYNAMICPROGRAMMING`.

[86] *Toll roads are spreading in America*, December 18, 2025. https://www.economist.com/united-states/2025/12/18/toll-roads-are-spreading-in-america

Let $G = (V, E)$ be a directed graph with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$ and *nonnegative integer edge costs* $\texttt{cost} : E \to \mathbb{Z}_{\geq 0}$, which model toll roads. We define the *cost* of a walk in $G$ as the sum of edge costs (with repetition) along that walk. Design and analyze an algorithm (the faster the better) that, given $k \in \mathbb{N}$, and $s, t \in V$, computes the length of the shortest $(s, t)$-walk that costs at most $k$. Your running may depend polynomially on $k$ (even though $k$ only takes $\log(k)$ bits to represent). (It might help to imagine $k \leq O(n)$.)

**Exercise K.208.** Prove or disprove: For any two events $A$ and $B$, if $\mathbf{P}[A] + \mathbf{P}[B] > 1$, then $\mathbf{P}[A \wedge B] > 0$.

**Exercise K.209.** Suppose you are a city planner, and you are worried that it takes too long for an ambulance to get from the university to the hospital. You've decided to build one new street to improve this particular route. You and your crack team have researched and gathered a list of potential streets that could be built; the task now is to choose one street that will improve this route the most.

To model this formally, let $G = (V, E)$ be a directed graph with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$, and let $s, t \in V$ be two vertices. Let $E$ be partitioned into two nonempty disjoint sets $E_0$ and $E_1$. $E_0$ represents the edges/streets that are "already built"; $E_1$ represents the edges/streets that you can add to the graph. Let $G_0 = (V, E_0)$ denote the graph of already built edges.

Consider the problem of identifying the single edge $e \in E_1$ so that, when you add $e$ to $G_0$, the distance from $s$ to $t$ in the augmented graph is as short as possible. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

As a follow up: What if you can add two edges? For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.210.** Let $G = (V, E)$ be a simple[87] directed graph where each edge is colored `red`, `white`, or `blue`. An American path is a path where the edge colors alternate `red`, `white`, `blue`, `red`, `white`, `blue`... (The first edge may be any color.) Consider the problem of computing the length of the longest American path in the graph. For this problem in each of the following settings, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard. You may assume that one can look up the color of a path in constant time.

[87] i.e., $G$ has no parallel edges.

1. $G$ is a DAG.
2. $G$ is a simple directed graph.

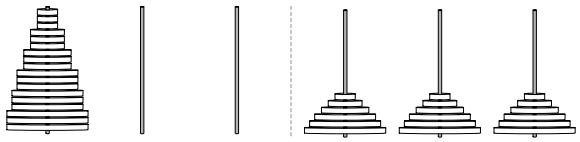
**Exercise K.211.** The racetrack problem in Chapter 5 of [Eri19].

**Exercise K.212.** For each of the following variations of the Tower of Hanoi problem, give an algorithm that solves the problem with the minimum number of ring moves. Recursive algorithms should have an explicit recursive spec. We will treat the recursive spec as an induction hypothesis to justify the correctness of the algorithm, and no explicit proof is otherwise required.[88] Non-recursive algorithms require a proof (which should be short).

You may assume as fact that the `Towers-of-Hanoi` algorithm (Fig. 2.1, on page 42) moves $n$ rings from one post to another in the minimum number of steps. Also, your algorithms are allowed to be mutually recursive (i.e., can depend on one another) across subproblems.

1. In the *easy towers of Hanoi* problem, we remove the restriction that the rings always have to be in sorted order on each post. The goal is still to move $n$ sorted rings from post $A$ to post $B$, in sorted order on post $B$.
2. In the *cyclic towers of Hanoi* problem, a ring can only be moved in "cyclic" order from post $A$ to post $B$, from post $B$ to post $C$, and from post $C$ to post $A$. The goal is to move $n$ sorted rings from post $A$ to post $B$.
3. The *double-cyclic towers of Hanoi* is like the cyclic towers of Hanoi problem except now the goal is to the $n$ sorted rings from post $A$ to post $C$.
4. In the *thick towers of Hanoi* problem, we have $3n$ rings of $n$ distinct sizes with three copies of each ring. (Rings of the same size can stack on top of each other.) The goal is to move all $3n$ rings from post $A$ to post $B$.
5. In the *triple towers of Hanoi* problem, we have $3n$ rings of $n$ distinct sizes with three copies of each ring. The goal is to distribute the rings so that each post has $n$ rings, one of each size, in sorted order.

[88] You may want to work out a full proof for yourself anyway for extra practice. Additional justification or comments might help the grader's understanding in the event of partial credit.



6. In the *American towers of Hanoi* problem, we have $n$ rings with each ring colored red, white or blue. The three posts are also colored red, white, and blue. Initially all the rings are stacked (in order of size) on the red post. The goal is to stack all the red rings on the red post, the blue rings on the blue post, and the white rings on the white post (again, in order of size).

7. In the *messy towers of Hanoi* problem, the $n$ rings are initially distributed arbitrarily over the three posts. (Each post is still in order.) The goal is to move all the rings to a single designated post.[89]

8. (Extra credit / just for fun) Invent a fun and interesting variant of the towers of Hanoi problem.

**Exercise K.213.** Word segmentation is the problem of dividing a string of written language into its component words. In English and many other languages using some form of the Latin alphabet, the space is a good approximation of a word divider (word delimiter), although this concept has limits because of the variability with which languages emically regard collocations and compounds. Many English compound nouns are variably written (for example, ice box = ice-box = icebox; pig sty = pig-sty = pigsty) with a corresponding variation in whether speakers think of them as noun phrases or single nouns; there are trends in how norms are set, such as that open compounds often tend eventually to solidify by widespread convention, but variation remains systemic. In contrast, German compound nouns show less orthographic variation, with solidification being a stronger norm. However, the equivalent to the word space character is not found in all written scripts, and without it word segmentation is a difficult problem.

Languages which do not have a trivial word segmentation process include Chinese, Japanese, where sentences but not words are delimited, Thai and Lao, where phrases and sentences but not words are delimited, and Vietnamese, where syllables but not words are delimited. In some writing systems however, such as the Ge'ez script used

[89]You are allowed to look at the current configuration and identify which post has the $n$th smallest ring, for any given $n$. You may have different cases depending on which post has the $n$th smallest ring.

for Amharic and Tigrinya among other languages, words are explicitly delimited (at least historically) with a non-whitespace character.

The Unicode Consortium has published a Standard Annex on Text Segmentation,[1] exploring the issues of segmentation in multiscript texts.[90]

We model the word segmentation problem as follows. Let $A[1..n]$ be a sequence of letters from a fixed alphabet. A *word segmentation* is a partition of $A$ into contiguous intervals $A[i_0 + 1..i_1], A[i_1 + 1..i_2], \ldots, A[i_\ell + 1..i_{\ell+1}]$; where $i_0 = 0$, $i_{\ell+1} = n$, and $\ell$ is any integer; such that each interval $A[i_j + 1..i_{j+1}]$ is a word. For example, the text

"youareawesome" has the word segmentation "you", "are", "awesome".

To determine which sequences form words, we assume access to a precomputed boolean array $\texttt{words}(1..n, 1..n)$, where for $i \leq j$, $\texttt{words}(i, j) = \texttt{true}$ iff $A[i..j]$ forms a word.

Given $A[1..n]$ and $\texttt{words}(1..n, 1..n)$, consider the problem of deciding if there is a word segmentation of $A$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.214.** Show that $\wedge$ and $\vee$ are not jointly associative by finding assignments $x, y, z \in \{0, 1\}$ such that $(x \vee y) \wedge z \neq x \vee (y \wedge z)$.

**Exercise K.215.** Each of the following problems describes a graph problem over an unweighted directed graph $G = (V, E)$ with $m$ edges and $n$ vertices. Each problem can be solved by constructing an auxiliary graph and calling a shortest path algorithm from this chapter as a black box. For each problem, (a) design an auxiliary graph and shortest path problem, (b) indicate which algorithm to apply and how, and (c) analyze the running time. No proof of correctness is required.

1. For two vertices $s$ and $t$, compute the length of the shortest walk from $s$ to $t$ with an even number of edges.[91]

2. For two vertices $s$ and $t$, compute the length of the shortest walk from $s$ to $t$ where the number of edges is a multiple of 5.

3. For two vertices $s$ and $t$, compute the length of the shortest walk from $s$ to $t$ where the number of edges is either a multiple of 2, or a multiple of 3.

4. For two vertices $s$ and $t$, compute the length of the shortest walk from $s$ to $t$ where the number of edges is either a multiple 3 or a multiple of 4, but not a multiple of 12.

[90] https://en.m.wikipedia.org/wiki/Text_segmentation#Word_segmentation

[91] For this and the other problems, if there is no such walk, then your algorithm should return $+\infty$.

5. For two vertices $s$ and $t$, compute the length of the shortest walk from $s$ to $t$ where the number of edges is of the form $5x + 8y$ for $x, y \in \mathbb{Z}_{\geq 0}$.

6. Suppose $G$ is a patriotic directed graph where all the edges are colored either red, white, or blue. An *American walk* is a walk where the edges alternate in color in the order of the American dream: red, white, blue, red, white, blue... Here the first edge could be any color and then the colors have to cycle through the American dream thereafter. Compute the length of the shortest American walk from $s$ to $t$.

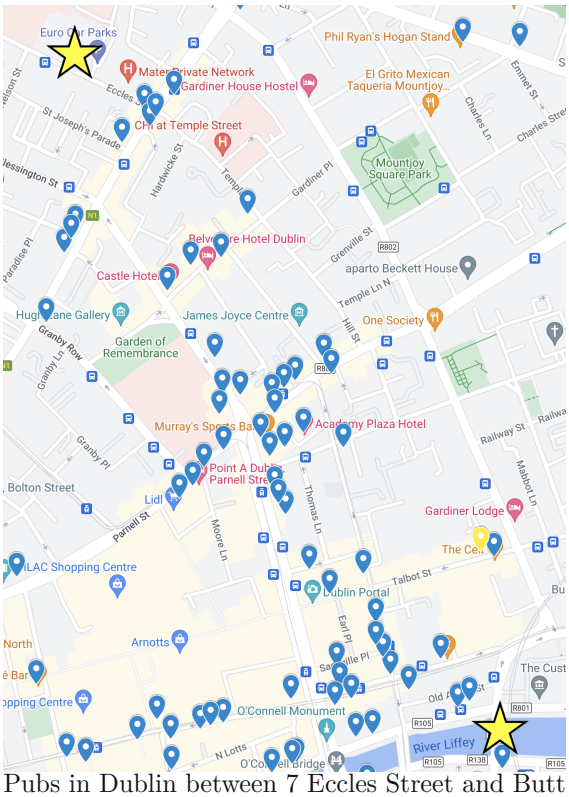
7. Suppose $G$ is again a patriotic directed graph where all the edges are colored either red, white, or blue. An *un-American walk* is a walk that never cycles through the American dream; that is, a walk where no three consecutive edges in the walk have colors that form any of the following three sequences: $(\text{red}, \text{white}, \text{blue})$, $(\text{white}, \text{blue}, \text{red})$, or $(\text{blue}, \text{red}, \text{white})$. Compute the length of the shortest un-American walk from $s$ to $t$.

8. Let $s, t \in V$ be two vertices. Consider the following game with two people Alice and Bob. Initially, Alice is on $s$, and Bob is on $t$. Each round, Alice and Bob are on two vertices, and they simultaneously take an outgoing edge to another vertex. The goal is to guide Alice and Bob to the same vertex with the minimum number of rounds or declare that it is impossible to have them meet. Compute the minimum number of rounds needed to have Alice and Bob meet at the same vertex.

**Exercise K.216.** Bloomsday is a commemoration and celebration of the life of Irish writer James Joyce, observed annually in Dublin and elsewhere on 16 June, the day his 1922 novel Ulysses takes place on a Thursday in 1904, and named after its protagonist Leopold Bloom.

This year for Bloomsday you've decided to celebrate with your own truly epic, special kind of bar crawl. You will start the crawl at Leopold Bloom's house at 7 Eccles Street. (The house was demolished in 1967, and the site is now occupied by the Mater Private Hospital.) You will end the crawl at Butt Bridge, by the Custom House, where the book ends.

Your goal is to get from 7 Eccles Street to the Butt Bridge as fast as possible, but there's a catch: You are not allowed to walk for more than 6.5 minutes without dropping into a pub for a drink. You might have to go out of your way to keep yourself properly hydrated. Moreover, every time you drop into a pub, you have to spend 30 minutes inside. Taking these requirements into account, what was originally a 23 minute walk (according to Google maps) may take all day after all.



Pubs in Dublin between 7 Eccles Street and Butt Bridge

Design and analyze an algorithm (the faster the better) that computes the infimum amount of time it takes to get from 7 Eccles Street to Butt Bridge according to the requirements above. (If it is impossible to do so, the algorithm should output $+\infty$). To model the map, let $G = (V, E)$ be a directed graph with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$. $\ell(e)$ represents the amount of time it takes to walk edge $e$. Let $s, t \in V$ represent 7 Eccles Street and the Butt Bridge, respectively. Let $P \subseteq V$ represent the pubs and let $k = |P|$. (Your running time may or may not depend on $k$, and we're focused on the regime where $k$ is much smaller than $n$.) The algorithm should compute the minimum amount of time it takes to get from $s$ to $t$ subject to the festive requirements described above.

**Exercise K.217.** According to a report from the World Shipping Council (WSC), 661 shipping containers (a.k.a. crates) were lost overboard in the year 2022. This is actually an improvement, as an average of 1,566 containers were lost at see each year in the fifteen year period from 2008 to 2022. (Cf. https://www.worldshipping.org/news/world-shipping-council-releases-containers-lost-at-sea-report-2023-update)

Now suppose you are running a crate shipping company. To try to minimize the number of crates that fall off the boat, you have decided to impose a strict weight limit $L \in \mathbb{N}$ on the total weight of shipping crates on any boat. Given a collection of $n$ crates of varying weight $W_1, \ldots, W_n \in \mathbb{N}$, your goal is to ship all $n$ crates with the minimum number of boats.

More formally, given the weight limit $L$ and the crate weights $W_1, \ldots, W_n$, the goal is to assign each crate to one of $k \in \mathbb{N}$ boats, for $k$ as small as possible, so that the total weight of containers assigned to any boat is at most $L$. (To be clear, $k$ is not part of the input; your job includes finding the minimum possible number of boats $k$. You can also assume that $L \geq W_i$ for all $i$.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

**Exercise K.218.** Let $G = (V, E)$ be a directed graph with $m$ edges and $n$ vertices, such that there is at least one directed path between every pair of vertices (in at least one direction or the other). Suppose each vertex $v \in V$ is labeled by a distinct integer $\pi(v) \in [n]$ (that is, $\pi : V \to [n]$ is injective). We say a pair of vertices $(u, v)$ is *reachably-misordered* if $u$ can reach $v$ and $\pi(v) < \pi(u)$.

Design and analyze an algorithm (as fast as possible) that counts the number of reachably-misordered pairs of vertices.

**Exercise K.219.** Recall the `quick-select` algorithm introduced in Section 25.1.3. The goal of this exercise is to prove that `quick-select` takes $O(n)$ time in expectation. Below we present two different approaches which offer two different perspectives. We ask you to submit solutions to just one of the two approaches, though we strongly recommending trying to solve and understanding both. Both analyses should use linearity of expectation and we ask you to point this out explicitly.

1. *Approach 1.* Analyze `quick-select` similarly to `quick-sort`, based on the sum of indicators $X_{ij}$.

One approach is to reduce to a separate analysis for each of the following 4 classes of pairs:

(a) $X_{ij}$ where $i < j < k$,

(b) $X_{ij}$ where $i < k < j$,

(c) $X_{ij}$ where $k < i < j$, and

(d) $X_{ij}$ where either $i = k$ or $j = k$.

For each case, show that the expected sum is $O(n)$.

2. *Approach 2.* The following approach can be interpreted as a randomized divide and conquer argument. We are arguing that with constant probability, we decrease the input by a constant factor, from which the fast (expected) running time follows.

   (a) Consider a coin that flips heads with probability $p$. Suppose we repeatedly flip the coin until it comes up heads. What is the expected number of coin tosses until it comes up heads? Prove your answer.[92]

   (b) Consider again `quick-select`. Consider a single iteration where we pick a pivot uniformly at random and throw out some elements. Prove that with some constant probability $p$, we either sample the $k$th element or throw out at least $1/4$ of the remaining elements.

   (c) For each integer $i$, prove that the expected number of iterations (i.e., rounds of choosing a pivot) of `quick-select`, where the number of elements remaining is in the range $[(4/3)^i, (4/3)^{i+1})$, is $O(1)$.

   (d) Fix an integer $i$, and consider the amount of time spent by `quick-select` while the number of elements remaining is greater than $(4/3)^{i-1}$ and at most $(4/3)^i$. Show that that the expected amount of time is $\leq O((4/3)^i)$

   (e) Finally, use the preceding part to show that the expected running time of `quick-select` is $O(n)$.

**Exercise K.220.** Let $G = (V, E)$ be a directed graph with real-valued edge lengths $\ell : E \to \mathbb{R}$, and let $s, t \in V$. Consider the problem of computing the length of the shortest path from $s$ to $t$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

[92] There is a slick proof.