Appendix H

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 1–8, in random order.

Exercise H.1. 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.[1]

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 H.2. 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.

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

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.

Exercise H.3. 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 H.4. 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 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. $D[i] \leftarrow A[k] + C[i]$ 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 H.1: A parallel prefix sum algorithm with $O(\log n)$ time and $O(n \log n)$ work.

> `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. H.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.

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.[2]

Remark H.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 H.5. 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 H.6. 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

[2]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 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?

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.[3]

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.

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

[3] For this and the other problems, if there is no such walk, then your algorithm should return $+\infty$.

the minimum number of rounds needed to have Alice and Bob meet at the same vertex.

Exercise H.7. 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 kth largest element among the combined elements of A and B. For simplicity you make assume that all the elements are distinct.[4]

1. Define a recursive spec for a recursive algorithm to solve this problem.
2. Show that if either m or n is ≤ 5 then we can find the kth element in $O(1)$-time by direct means. (This is to help declutter your implementation and avoid fencepost errors.)
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 H.8. 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:

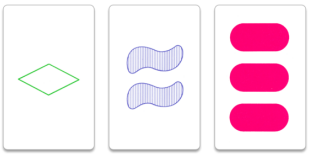
1. The shape, which is either a diamond, a squiggle, or an oval.
2. The number of shapes, which is either 1, 2, or 3.
3. The color of the shape, which is either red, green, purple.
4. The shading of the shape, which is either solid, striped, or open.

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.

[4] 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.



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 $C[1..p]$.[5] 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 H.9. 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.

[5] 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. 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

$$r(e_1) + \alpha r(e_2) + \cdots + \alpha^{k-1} r(e_k).$$

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 H.10. 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 H.11. 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 $n - 1$ vertices.

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.[6]

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.

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

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.

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 H.12. 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 H.13. 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 H.14. 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 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.[7]


Peter, Egon, and Ray

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.

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

[7] I can only assume that *noone* has seen the original Ghostbusters movie, since it's even older than Good Will Hunting...

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 H.15. 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.[8]

Exercise H.16. 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[9] 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 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.

[8]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.

[9]or paragraphs, or more... sigh...

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 H.17. 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.
4. A *Latin square* is an $n \times n$ grid filled with n symbols such that each symbol appears exactly onece in each row and column. A *partial Latin square* has some cells filled (obeying the Latin square constraints) and others empty. Prove that any partial Latin square can be completed to a Latin square.

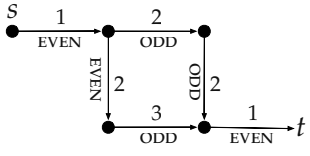
Exercise H.18. Red light, green light! The following model is inspired by streetlights which, at a given point in time, allow traffic to enter some streets and block traffic from entering others.

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 H.19. 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 H.20. 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.[10]

Exercise H.21. 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.

[10] *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 H.43 before this one.

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 Δ 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 $\chi_{\text{L}}(K_{n,n}) \geq \log_2 n$.
6. Prove the $\chi_{\text{L}}(K_{n,n}) \leq O(\log n)$. (This is tricker; *randomization* might be helpful.)

Exercise H.22. Prove or disprove: every walk in a directed acyclic graph has finite length.

Exercise H.23. For each of the recursive specifications below, design a recursive algorithm implementing the specification. (No proof or analysis is needed.)[11]

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.
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.)

[11] 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 58.

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.

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.[12]

Exercise H.24. 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

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

(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 H.25. 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 H.26. 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 kth 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 H.27. 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 H.20 on page 630.)

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 H.28. 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 α, β, γ reflect prior assumptions on the likelihood of each operation.

Exercise H.29. Let $A[1..m]$ and $B[1..n]$ be two arrays. Each of the following problems asks for some aspect (e.g., the length) of a common subsequence of $A[1..n]$ and $B[1..n]$ satisfying or optimizing certain properties. (A common subsequence of A and B is a sequence that is a subsequence of both A and B.) Design and analyze an algorithm for each of these problems, addressing Items 1–5 from Section 5.2.[23]

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 consists 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 consists of integers. Compute the length of the longest common convex subsequence of A and B. (Cf. problem 2)

Exercise H.30. Let $p(x,y) = \sum_{i=0}^{m}\sum_{j=0}^{n} a_{ij}x^iy^j$ and $q(x,y) = \sum_{i=0}^{m}\sum_{j=0}^{n} b_{ij}x^iy^j$ 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 $O(m^2n^2)$ time with nested loops. Design and analyze a faster algorithm computing the product r.

Exercise H.31. 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 α 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 H.32. 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:

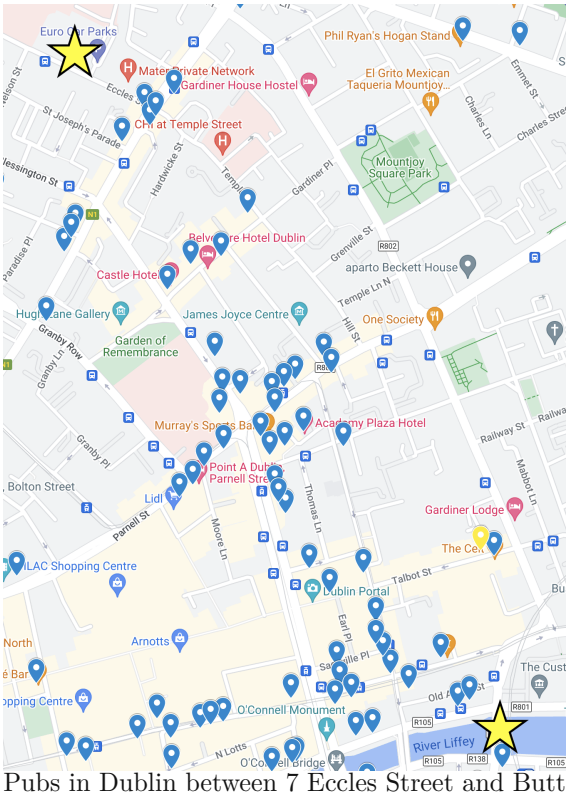
> `reach(`s`,`t`,`k`)`: Given two vertices $s, t \in V$ and $k \in \mathbb{N}$, returns true if s can reach t in at most k edges.

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

Exercise H.33. 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.

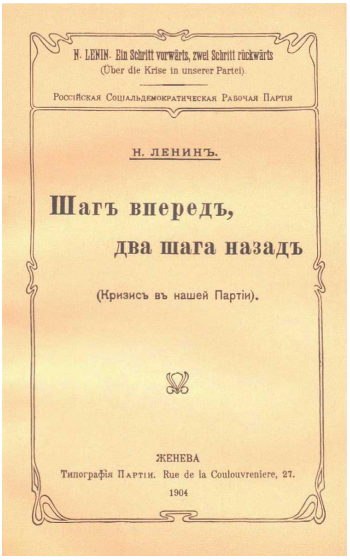


Pubs in Dublin between 7 Eccles Street and Butt Bridge

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 H.34. *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.


N. LENIN. Ein Schritt vorwärts, zwei Schritt rückwärts
(Über die Krise in unserer Partei).
Российская Социальдемократическая Рабочая Партія
Н. ЛЕНИНЪ.
Шагъ впередъ,
два шага назадъ
(Кризисъ въ нашей Партіи).
ЖЕНЕВА
Типографія Партіи. Rue de la Coulouvreniere, 27.
1904

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 H.35. 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 H.36. 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[13] 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 H.37. 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.[14] You may assume that all the integers in X are distinct (amongst themselves), and that all the integers in Y are distinct (amongst themselves).[15]

[13] 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`.

[14] That is, for fixed z, there is only one choice of $x \in X$ and one choice of $y \in Y$ such that $z = x + y$.

[15] 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.

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.[16]

3. (2 pt.) Suppose we now that 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.[17]

Hint: For part 2, try "changing the input" to an algorithm from Section 4.3 (that is not the FFT, at least directly).

Exercise H.38. 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 H.39. 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

[16] It might be helpful to work through very simple examples such as $X = Y = \{1\}$ and $X = Y = \{0, 1\}$.

[17] It might help to work through $k = 4$ to build your intuition.

explicit proof is otherwise required.[18] 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 41) moves n rings from one post to another in the minimum number of steps.

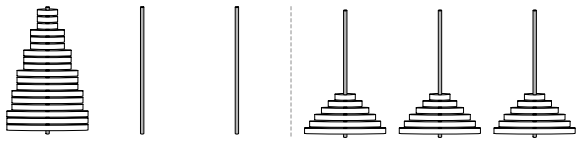
1. In the *easy towers of Hanoi* problem, we remove the restriction that the rings always have to be in sorted order on each post. The goal is still to move n sorted rings from post A to post B, in sorted order on post B.

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.



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).

[18] 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.

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.[19]

8. (Extra credit) Invent a fun and interesting variant of the towers of Hanoi problem.

Exercise H.40. 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 H.41. 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 H.42. 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 $R(s, t) \leq \binom{s+t-2}{s-1}$.

[19] You are allowed to look at the current configuration and identify which post has the nth smallest ring, for any given n. You may have different cases depending on which post has the nth smallest ring.

5. Prove Ramsey's theorem.

6. Prove that $R(k,k) > 2^{k/2}$. (This is trickier; *randomization* might be helpful.)

Exercise H.43. 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 H.44. Recall that a sequence of numbers $x_1, \ldots, x_k$ is *increasing* if $x_1 \leq x_2 \leq \cdots \leq x_k$. Here we are interested in sequences that are increasing, but don't grow too fast, in the following sense.

Recall that the Fibonacci sequence

$$1, 1, 2, 3, 5, 8, ...$$

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.

Let $A[1..n]$ be an array of numbers. Design and analyze an algorithm that computes the length of the longest increasing and sub-Fibonacci subsequence of A.

Exercise H.45. Let P be a set of n points in $\mathbb{R}^2$.[20] 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?)

Exercise H.46. 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.

[20] 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 H.47. 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.

Exercise H.48. 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.

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 H.49. 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 kth 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 H.50. Problem 6 of exercise H.54 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.

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

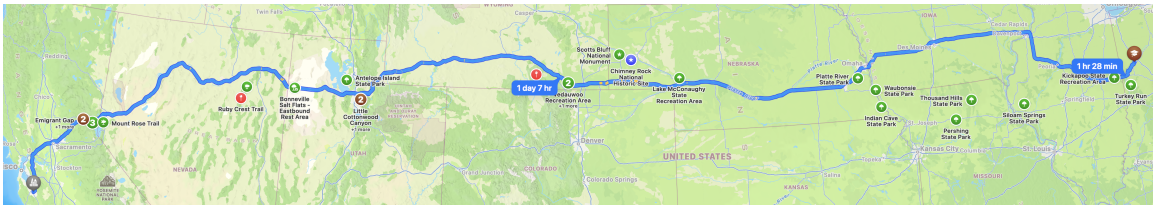
Above, `racecar` is a palindrome subsequence, with total weight $22 = 5 + 3 + 7 + 2 + 4 + 5 - 1$.

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 H.51. 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 ith 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 H.52. 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 $1, \dots, n$ 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 H.53. 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 Δ-regular bipartite graph. (Every vertex has degree Δ). Prove that $\xi(G) = \Delta$.[21]
3. Now let G be a bipartite graph with maximum degree Δ. Prove that $\xi(G) = \Delta$. (This is called *König's theorem.*)

Exercise H.54. Below is a series of optimization problems that takes as input an array $A[1..n]$ of integers, and asks for optimal subsequences of A satisfying certain properties.[22] Design and analyze an algorithm for each of these problems, addressing Items 1–5 from Section 5.2.[23]

[21] Hint: you might want to do exercise H.17 first.

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

[23] 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 the rest of the details ourselves. That said, some additional commentary may help the grader, especially in the event of partial credit.

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

5. 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.)

6. 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.

7. Compute the length of the shortest supersequence of A that is a palindrome.[24]

8. 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.)

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

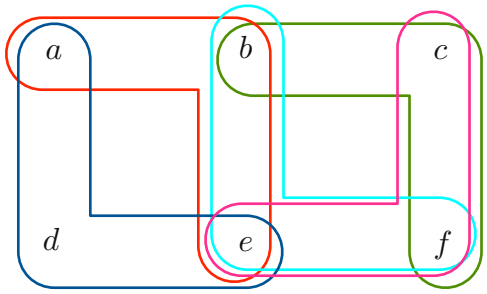
9. (Just for fun) Invent your own fun and interesting version of a dynamic programming problem over A.

Exercise H.55. 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.

Exercise H.56. 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 H.57. Here we assume the same model as exercise H.66 – an array $A[1..n]$ of incomparable elements[25] – 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.[26]

Exercise H.58. Prove that every directed acyclic graph has at least one source, and at least one sink.

Exercise H.59. 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.

[25] In addition to disallowing sorting, no hash tables are allowed as well.

[26] Your algorithm should be faster than $O(n^2)$ (which is the trivial running time) for, e.g., $k = n^{1/3}$.

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 H.60. 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 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.[27]

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 ℓ 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 H.61. Problems 1–5 describe a search problem on a nonempty array $A[1..n]$ of comparable elements.[28] For each of these problems, design a recursive algorithm that solves the search problem as fast as possible.[29] Briefly analyze the running time of each algorithm.[30]

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.

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

[27] https://en.m.wikipedia.org/wiki/Text_segmentation#Word_segmentation

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

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

[30] 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.

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) Invent your own fun and interesting search problem on an array of comparable elements.

Exercise H.62. 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 H.63. 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 H.64. 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).

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 H.65. The racetrack problem in Chapter 5 of [Eri19].

Exercise H.66. 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 $O(n \log n)$.[31]

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

4. Analyze the running time of your algorithm.

Exercise H.67. Let $G = (V, E)$. 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.

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

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

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

Exercise H.68. 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 H.69. 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 H.70. 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 H.70?