Standard homework rubric. All homework problems are graded on a 4-point scale:

0. Missing, meaningless, or Deadly Sin.
1. Meaningful but incorrect, or "I don't know".
2. Based on the right ideas, but with major errors or omissions.
3. Mostly correct, with a few minor errors or omissions.
4. Absolutely perfect. (This score should be rare.)

(Note to graders: ***All*** scores ***must*** be justified by narrative feedback.)

If a problem has multiple parts, each part is graded separately on this 4-point scale. Unless specified otherwise, the overall score for the problem is a simple average of the part scores.

For many homework problems, the solution will include a more detailed 10-point rubric, which reflects how we would grade that problem on an exam. As stated in the course policies, a score of x on this 10-point scale is equivalent to a score of $\lceil x/3 \rceil$ on the actual 4-point homework scale.

1. (a) Describe an algorithm to sort an arbitrary stack of n pancakes, which uses as few flips as possible in the worst case. *Exactly* how many flips does your algorithm perform in the worst case?

Solution: Here is an algorithm that resembles selection sort. The stack of pancakes is represented by an array $P[1,..n]$, where $P[i]$ is the size of the ith pancake from the top of the stack. The algorithm uses a subroutine FLIP(k) that flips the top k pancakes.

```
FLIPSORT(P[1..n]):
  for i ← n down to 3
      k ← position of the ith smallest pancake
      FLIP(k)        ⟨⟨Flip it to the top⟩⟩
      FLIP(i)        ⟨⟨Flip it into place⟩⟩
  if P[1] > P[2]
      FLIP(2)
```

The algorithm uses at most $\mathbf{2n-3}$ flips in the worst case. (The time to compute k in the second line is completely immaterial; the problem only asked for the worst-case number of flips.) ■

(b) Now suppose one side of each pancake is burned. Describe an algorithm to sort an arbitrary stack of n pancakes, so that the burned side of every pancake is facing down, using as few flips as possible in the worst case. *Exactly* how many flips does your algorithm perform in the worst case?

Solution: We modify the previous algorithm slightly. Whenever each pancake reaches the top of the stack, we flip it if necessary to ensure that its burned side is *up*, so that when we flip it down to its proper place, its burned side is down.

```
BURNEDFLIPSORT(P[1..n]):
  for i ← n down to 2
      k ← position of the ith smallest pancake
      FLIP(k)        ⟨⟨Flip it to the top⟩⟩
      if the top pancake's burned side is down
          FLIP(1)
      FLIP(i)        ⟨⟨Flip it into place⟩⟩
  if the top pancake's burned side is up
      FLIP(1)
```

The algorithm uses at most $\mathbf{3n-2}$ flips in the worst case. ■

2. Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution.

Solution: Let $M[1..n, 1..n]$ be the input maze. We construct a directed graph $G = (V, E)$ whose vertices correspond to grid cells and whose edges correspond to legal moves. More formally, we define the vertices and edges as follows:

$$V := \{(i, j) \mid 1 \le i, j \le n\}$$

$$E := \{(i, j) \to (i, k) \mid |j - k| = M[i, j]\} \cup \{(i, j) \to (k, j) \mid |i - k| = M[i, j]\}.$$

Any sequence of legal moves in M corresponds to a directed path in G. Thus, we need to find the shortest path from $(1, 1)$ to (n, n) in G. We can compute this shortest path using breadth-first search in $O(V + E) = O(n^2)$ time. ■

Standard rubric for all graph-reduction problems.
- 2 points for correct vertices
- 2 points for correct edges
  - –½ for not explicitly stating that edges are directed
  - –1 for explicitly describing undirected edges
- 2 points for stating the correct problem (shortest paths)
  - – "Breadth-first search" is not a problem; it's an algorithm.
- 2 points for correctly applying the correct algorithm (breadth-first search)
  - –1 for using Dijkstra instead of BFS
- 2 points for time analysis in terms of n.

3. (a) Prove that every non-negative integer can be written as the sum of distinct, non-consecutive Fibonacci numbers.

Solution: Let n be an arbitrary non-negative integer. Assume that every non-negative integer less than n is the sum of distinct nonconsecutive Fibonacci numbers. There are two cases to consider.

- If $n = 0$, then n is equal to the empty sum. The empty set $\emptyset$ does not contain two consecutive Fibonacci numbers.
- Suppose $n \geq 1$. Let F_k be the largest Fibonacci number such that $F_k \leq n < F_{k+1}$, and let $m = n - F_k$. Because $n \geq 1$, we have $m < n$, and because $n < F_{k+1}$, we have $m < F_{k+1} - F_k = F_{k-1}$. The inductive hypothesis implies that m is the sum of nonconsecutive Fibonacci numbers, each of which is at most m and therefore less than F_{k-1}. Thus, adding F_k to this sum gives us a sum of distinct, non-consecutive Fibonacci numbers whose value is n.

In all cases, we conclude that n is the sum of distinct nonconsecutive Fibonacci numbers. ■

(b) Prove that $F_{-n} = -F_n$ if and only if n is even.

Solution (induction): It's actually easier to prove the following stronger statement:

$$F_{-n} = \begin{cases} -F_n & \text{if } n \text{ is even,} \\ F_n & \text{if } n \text{ is odd.} \end{cases}$$

By symmetry, it suffices to prove this statement for all *non-negative* integers n.

Let n be an arbitrary non-negative integer. Assume for all integers m such that $|m| < |n|$, that $F_{-m} = -F_m$ if m is even and $F_{-m} = F_m$ if m is odd. There are four cases to consider:

- If $n = 0$, then n is even and $F_{-0} = 0 = -F_0$.
- If $n = 1$, then n is odd and $F_{-1} = 1 = F_1$.
- Suppose $n > 1$ and n is even. Then $n-2$ is even and $n-1$ is odd, so the inductive hypothesis implies that $F_{-n+2} = -F_{n-2}$ and $F_{-n+1} = F_{n-1}$. It follows that

$$F_{-n} = F_{-n+2} - F_{-n+1} = -F_{n-2} - F_{n-1} = -F_n.$$

- Finally, suppose $n > 1$ and n is odd. Then $n-2$ is odd and $n-1$ is even, so the inductive hypothesis implies that $F_{-n+2} = F_{n-2}$ and $F_{-n+1} = -F_{n-1}$. it follows that

$$F_{-n} = F_{-n+2} - F_{-n+1} = F_{n-2} + F_{n-1} = F_n.$$

In all cases, we conclude that $F_{-n} = -n$ if n is even and $F_{-n} = n$ if n is odd. ■

(c) Prove that *every* integer—positive, negative, or zero—can be written as the sum of distinct, non-consecutive Fibonacci numbers *with negative indices*.

Solution: I'll actually prove the following stronger statement: Every integer n can be written as the sum of distinct non-consecutive Fibonacci numbers with negative indices such that for all k, the following properties hold:

- If $0 < n \le F_{2k}$, then every index in the sum is at least $-(2k-1)$.
- If $-F_{2k+1} < n < 0$, then every index in the sum is at least $-2k$.

To simplify the proof, I will refer to a sum of negative-index Fibonacci numbers with these properties as a ***legal sum of Fibonacci numbers***.

Let n be an arbitrary integer. Assume that every integer m such that $|m| < |n|$ can be written as a legal sum of Fibonacci numbers. There are three cases to consider:

- The base case $n = 0$ is vacuously satisfied by the empty sum.
- Suppose $n > 0$. Let k be the unique positive integer such that

  $$F_{2k-2} < n \le F_{2k},$$

  and let $m = n - F_{-(2k-1)} = n - F_{2k-1}$ (by part (b)). The Fibonacci recurrence implies

  $$-F_{2k-3} < m \le F_{2k-2}$$

  The inductive hypothesis implies that we can write m as a legal sum of Fibonacci numbers; in particular:
  - If $m = 0$, the sum is empty.
  - If $m > 0$, every index in the sum is at least $-(2k-3)$.
  - If $m < 0$, every index in the sum is at least $-(2k-4)$.

  In all cases, neither $F_{-(2k-1)}$ nor $F_{-(2k-2)}$ appears in the sum for m. Thus, adding $F_{-(2k-1)}$ to the sum for m gives us a legal sum of Fibonacci numbers for n.
- Suppose $n < 0$. Let k be the unique positive integer such that

  $$-F_{2k+1} < n \le -F_{2k-1},$$

  and let $m = n - F_{-2k} = n + F_{2k}$ (by part (b)). The Fibonacci recurrence implies

  $$-F_{2k-1} < m \le F_{2k-2}$$

  The inductive hypothesis implies that we can write m as a legal sum of Fibonacci numbers; in particular:
  - If $m = 0$, the sum is empty.
  - If $m > 0$, every index in the sum is at least $-(2k-3)$.
  - If $m < 0$, every index in the sum is at least $-(2k-2)$.

  In all cases, neither $F_{-(2k-1)}$ nor $F_{-(2k)}$ appears in the sum for m. Thus, adding F_{-2k} to the sum for m gives us a legal sum of Fibonacci numbers for n.

In all cases, we conclude that n is the sum of distinct non-consecutive Fibonacci numbers with negative indices. ■

Standard rubric for all induction proofs.

- 1 point for explicitly considering an **arbitrary** element of the domain of the theorem. (Unnecessary if the rest of the proof is correct.)
- 1 point for stating a valid induction hypothesis equivalent to "Assume there is no smaller counterexample."
  - **Ambiguous** induction hypotheses like "Assume the statement is true for all $k < n$." are not valid. *What* statement? The theorem you're trying to prove doesn't use the variable k, so that can't possibly be the statement you mean.
  - **Meaningless** induction hypotheses like "Assume that k is true for all $k < n$" are not valid. Only propositions can be true or false; k is an integer, not a proposition.
  - **False** induction hypotheses like "Assume that $k < n$ for all k" are not valid. The inequality $k < n$ does *not* hold for all k, because it does not hold when $k = n + 5$.
  - **Implicit** induction hypotheses like "The proof is by induction on $|n|$." are valid if and only if the rest of the proof is correct.
  - By course policy, stating a **weak** inductive hypothesis triggers an automatic zero, unless the proof is otherwise *perfect*.
- 1 point for explicit and clearly exhaustive case analysis.
  - No penalty for overlapping or redundant cases. However, mistakes in redundant cases are still penalized.
- 2 points for all base cases
- 2 points for correctly applying the *stated* inductive hypothesis.
  - It is not possible to correctly apply an invalid inductive hypothesis.
  - It is possible to incorrectly apply a valid inductive hypothesis.
- 3 points for other details of the inductive cases
  - In particular, the conditions under which each inductive argument works must match the conditions of the case analysis.

In particular, an otherwise perfect proof that does not state an inductive hypothesis (even implicitly) is worth 9/10. An otherwise perfect proof that states one valid inductive hypothesis but actually uses something else is worth 8/10. An otherwise perfect proof that states an invalid inductive hypothesis is worth 7/10.

4. ***[Extra credit]*** Describe and analyze an algorithm to transform an *arbitrary* binary tree T with distinct node values into a binary search tree, using only rotations and swaps.

Solution (exchange nodes): I'll describe an algorithm to sort an arbitrary n-node binary tree with distinct node values using $O(n \log n)$ operations. Without loss of generality, assume the values in the nodes are the integers 1 through n.

```
TreeSort(T):
  make T perfectly balanced
  for i ← 1 to n
      exchange the node with value i with the node in position i
```

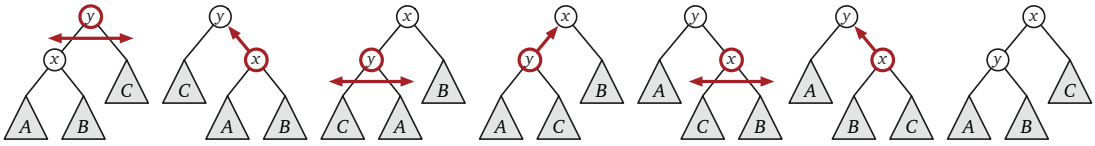
In the preprocessing phase, we perform $O(n)$ rotations to make the T perfectly balanced as possible. Specifically, we first change T to a path of left children as follows: While any node on the left spine of T has a right child, rotate that child to the left. Each left rotation increases the length of the left spine by 1, so the while loop ends after at most $n-1$ rotations. By running the same algorithm backward in time, we can transform the leftward path into a perfectly balanced tree in at most $n-1$ more rotations.

To complete the description of the algorithm, we need to describe how to swap two arbitrary nodes (just the nodes, not their subtrees) in a balanced binary tree using at most $O(\log n)$ operations.

> My original solution was based on an algorithm to swap **siblings** using only 13 rotations and swaps, but the following algorithm, which was submitted by several students, is significantly simpler and more efficient!

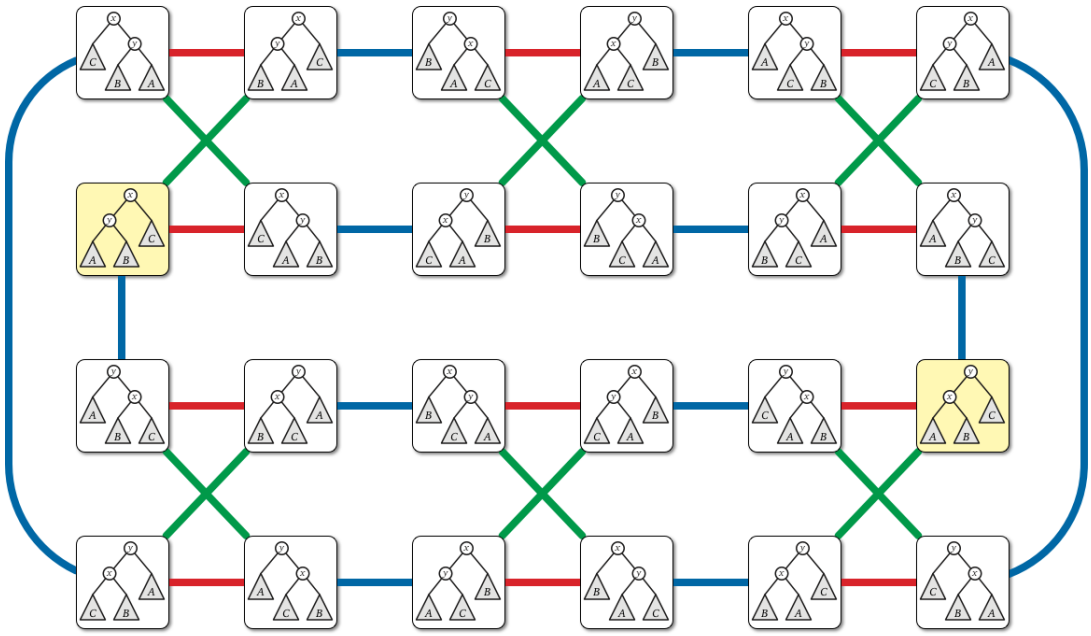
We can swap any node in T with its parent using three swaps and three rotations, as follows. (In fact, there are several different sequences of six operations that will perform this exchange; see the graph on the next page.)

```
ExchangeWithParent(x):
  y ← parent(x)        ⟨⟨ (AxB)yC ⟩⟩
  Swap(y)              ⟨⟨ Cy(AxB) ⟩⟩
  Rotate(x)            ⟨⟨ (CyA)xB ⟩⟩
  Swap(y)              ⟨⟨ (AyC)xB ⟩⟩
  Rotate(y)            ⟨⟨ Ay(CxB) ⟩⟩
  Swap(x)              ⟨⟨ Ay(BxC) ⟩⟩
  Rotate(x)            ⟨⟨ (AyB)xC ⟩⟩
```



Exchanging a node with its parent.

Now consider arbitrary simple path in T through nodes $w_0, w_1, w_2, \ldots, w_\ell$, where each node w_i is either the parent or child of its successor w_{i+1}. We can exchange the endpoints w_0 and w_ℓ of this path as follows. First, for each i from 1 to ℓ, swap w_0 with w_i. Then, for each i in decreasing order from $\ell-1$ to 1, swap w_ℓ with w_i. The total number of node-swaps is $2\ell-1$, and therefore the total number of primitive operations is $12\ell-6$. Finally, because



The graph of all possible configurations of two adjacent nodes in a node-labeled binary tree.
Blue edges indicate rotations. Red edges indicate swaps at the parent; green edges indicate swaps at the child.

we balanced T in the preprocessing phase, any two nodes in T are connected by a path of length $O(\log n)$. It follows that we can swap any two nodes in T in $O(\log n)$ steps, as required.

Alternatively, we can swap any two nodes x and y in T using $O(\log n)$ rotations and only $O(1)$ subtree swaps as follows, assuming without loss of generality that x is not a descendant of y. Rotate x upward until it is an ancestor of y; rotate y upward until it is a child of x; swap x and y; and finally, undo all the rotations that brought x and y together. Using this second strategy to swap nodes, TREESORT uses a total of ***$O(n \log n)$ rotations and $O(n)$ swaps*** in the worst case. ■

Solution (exploit treap; sketch): The following alternative solution is arguably simpler, but exploits a randomized binary tree structure, called a *treap*, that we won't see until later in the course.

Without loss of generality, assume the values in the nodes are the integers 1 through n. We associate two additional values with each node in the input tree T: its *rank*, which is its position in an inorder traversal of T, and its *priority*, which is a random number generated uniformly from the interval $[0, 1]$. (Alternatively, we can assign ranks using a random permutation of the integers 1 through n.)

First, in a preprocessing phase, we perform rotations so that the priorities obey the heap property; we require the priority of every node is smaller than the priorties of its children. We can change and tree into any other with the same inorder node sequence in $O(n)$ rotations. The resulting tree T is a treap with respect to the ***ranks*** and priorities. With high probability, every node in T has depth $O(\log n)$; see the treap notes for details.

Next, move the node v with the smallest *value* to the far left of the tree, and then to the root, as follows.

- First, for all ancestors u of v in order of increasing depth, if u is a right child, then swap the subtrees of the parent of v. These swaps move v to the left spine. The number of swaps is at most the depth of v, which is $O(\log n)$ with high probability.
- Next, as long as v has a left child, rotate that child upward (making v that node's right child) and then swap its subtrees. These two operations keep v on the left spine and decrease the number of v's descendants. The total number of operations is at most twice the length of the left spine, which is $O(\log n)$ with high probability.
- Finally, rotate v upward until it is the root of the tree. Again, the number of rotations is at most the length of the left spine, which is $O(\log n)$ with high probability.

Once v is the root, its left subtree is empty and its right subtree is still a treap for the remaining nodes. If the right subtree is empty, we are done. Otherwise, we recursively convert the right subtree into a binary search tree.

With high probability, the entire algorithm uses at most $\boldsymbol{O(n \log n)}$ ***rotations and swaps***. (If necessary, we can actually compute priorities such that the algorithm uses at most $O(n \log n)$ rotations and swaps in the worst case.) The final binary search tree is in fact just a sorted linked list of right children; we can balance this tree with $O(n)$ additional rotations if necessary. ■

Extra credit policy. We will generally grade extra credit work by a stricter standard than required homework problems. In particular, the "I don't know" policy does not apply to extra credit problems; partial credit requires at least a sketch of a correct solution.

Extra credit points will be added to your overall score ***after*** grade cutoffs have been computed, so they can only help you. However, extra credit points are not necessarily worth the same as standard homework points.