General rules for homework in this class:

- Teams of up to three people may submit joint homework solutions. You may use any resource at your disposal—printed, electronic, human, or other—but cite/acknowledge your sources, exactly as if you were writing a research paper. In particular, you are welcome to discuss the problems in larger groups, but each team of three must write up their own solutions in their own words.
- Each team should submit solutions for three problems. Some homework sets (like this one) will have more than three problems to choose from.
- Please typeset your solutions, preferably using LaTeX. Source files for the homework questions are available on the course web site. Complete solutions for each numbered problem (excluding starred subproblems) should take at most two typeset pages.
- *Stars indicate problems that I don't know how to solve; these problems may or may not actually be open, or even interesting. ★Larger stars indicate problems that I *know* are open (but still not necessarily interesting). I don't expect you to solve these problems, but I will give credit for any interesting observations, partial results, or creative failures.

Homework 1

1. Consider the following variant of the Bentley-Saxe logarithmic construction. Instead of keeping a sequence of static data structures whose sizes are distinct powers of two, the component structure sizes are distinct *Fibonacci numbers*, defined by the standard recurrence $F_n = F_{n-1} + F_{n-2}$ with bases cases $F(0) = 0$ and $F(1) = 1$. Here is the new insertion algorithm:

```
INSERT(x):
  if D_2 = NULL
      i ← 2
  else
      i ← 1
  D_i ← new data structure for {x}
  i ← i + 2
  while D_{i-1} ≠ NULL and D_{i-2} ≠ NULL
      D_i ← new data structure for D_{i-1} ∪ D_{i-2}
      delete D_{i-1}
      delete D_{i-2}
      i ← i + 2
```

   (a) Prove that INSERT is actually correct! How do we know D_i doesn't already point to a data structure in the first line of the while loop?

   (b) Suppose the time to construct a new static data structure of size n takes $P(n)$ time. Prove that the amortized running time of INSERT is $O(P(n)/n)$.

   (c) Describe a lazy version of this data structure where the *worst-case* running time for INSERT is $O(P(n)/n)$, similar to Overmars and van Leuuwen's lazy version of Bentley-Saxe. *[Hint: First design a lazy Fibonacci counter!]*

2. After the Great Academic Meltdown of 2020, you get a job as a cook's assistant at Jumpin' Jack's Flapjack Stack Shack, which sells arbitrarily tall stacks of pancakes for just four bits (50 cents) each. Jumpin' Jack insists on giving his customers sorted stacks of pancakes, with smaller pancakes on top of larger pancakes. Also, whenever a pancake goes to a customer, its top side must not be burned.

   The cook provides you with a unsorted stack of n perfectly circular pancakes, of n different sizes, possibly burned on one or both sides. Your task is to throw out the pancakes that are burned on both sides (and *only* those) and sort the remaining pancakes so that their burned sides (if any) face down. Your only tool is a spatula. You can insert the spatula under any pancake and then either *flip* or *discard* the stack of pancakes above the spatula.

   More concretely, let us represent a stack of pancakes by a sequence of distinct integers between 1 and n, representing the sizes of the pancakes, with each number marked to indicate the burned side(s) of the corresponding pancake. For example, $\underline{1}\,\overline{4}\,3\,\overline{\underline{2}}$ represents a stack of four pancakes: a one-inch pancake burned on the bottom; a four-inch pancake burned on the top; an unburned three-inch pancake, and a two-inch pancake burned on both sides. We store this sequence in a data structure that supports the following operations:

   - Position(x): Return the position of integer x in the current sequence, or 0 if x is not in the sequence.
   - Value(k): Return the kth integer in the current sequence, or 0 if the sequence has no kth element. Value is essentially the inverse of Position.
   - TopBurned(k): Return True if and only if the top side of the kth pancake in the current sequence is burned.
   - Flip(k): Reverse the order and the burn marks of the first k elements of the sequence.
   - Discard(k): Discard the first k elements of the sequence.

   (a) Describe an algorithm to filter and sort any stack of n burned pancakes using $O(n)$ of the operations listed above. Try to make the big-Oh constant small.

   $$\underline{1}\,\overline{4}\,3\,\overline{\underline{2}} \xrightarrow{\text{Flip}(4)} \overline{\underline{2}}\,3\,\underline{4}\,\overline{1} \xrightarrow{\text{Discard}(1)} 3\,\underline{4}\,\overline{1} \xrightarrow{\text{Flip}(2)} \overline{4}\,3\,\overline{1} \xrightarrow{\text{Flip}(3)} \underline{1}\,3\,\underline{4}$$

   (b) Describe a data structure that supports each of the operations listed above in $O(\log n)$ amortized time. Together with part (a), such a data structure gives us an algorithm to filter and sort any stack of n burned pancakes in $O(n \log n)$ time.

   ⋆(c) Prove or disprove: Sorting *unburned* pancakes requires at least $\Omega(\log n)$ amortized time per operation. More concretely: Any data structure that stores a permutation of the set $\{1, 2, \ldots, n\}$ and supports the operations Position, Value, and Flip requires at least $\Omega(\log n)$ amortized time per operation in the cell-probe model. *[We'll talk about cell-probe lower bounds later in the semester. Meanwhile, see Erik Demaine and Mihai Pătraşcu's STOC 2004 paper "Lower Bounds for Dynamic Connectivity".]*

   ★(d) Describe an integer-RAM algorithm that computes, in $o(n \log n)$ time, a sequence of $O(n)$ Flip operations that sorts a given stack of n unburned pancakes. Alternatively, prove that no such algorithm exists. *[Hint: There are sorting algorithms that run in $o(n \log n)$ time on the integer RAM; that's not the hard part.]*

3. In class we described a version of Sleator and Tarjan's link-cut trees that supports *path operations* via lazy propagation. Specifically, the data structure maintains a *value* at each node in a dynamic rooted forest, and supports the following operations in $O(\log n)$ amortized time, in addition to the core operations LINK, CUT, and FINDROOT:

   - GETVALUE(v): Return the value associated with v.
   - SETVALUE(v, x): Change the value associated with v to x.
   - MINPATH(v): Return the *ancestor* of v with minimum value.
   - ADDPATH(v, Δ): Add Δ to the value of every *ancestor* of v.

   Extend link-cut trees further, to support the following *subtree operations*:

   - MINSUBTREE(v): Return the *descendant* of v with minimum value.
   - ADDSUBTREE(v, Δ): Add Δ to the value of every *descendant* of v.

   The algorithms for LINK, CUT, FINDROOT, GETVALUE, SETVALUE, MINPATH, and ADDPATH should require only *minor* modifications; all operations should run in $O(\log n)$ amortized time. You may assume that every node in the *represented* forest has degree at most 3. (This assumption is not actually necessary, but it dramatically simplifies the solution.)

4. Demaine *et al.* describe the following three greedy algorithms in their paper "The geometry of binary search trees". The input to each algorithm is an array set X of m integers between 0 and $n-1$; each array element $X[i]$ represents the point $(X[i], i)$. Each algorithm outputs a superset Y of the point set represented by X.

```
GREEDY◿(X[1..m]):
  Y ← ∅
  for i ← 1 to m
      add (X[i], i) to Y
      SATISFY◿(Y, X[i], i)
  return Y
```

```
GREEDY⊠(X[1..m]):
  Y ← ∅
  for i ← 1 to m
      add (X[i], i) to Y
      SATISFY◿(Y, X[i], i)
      SATISFY◹(Y, X[i], i)
  return Y
```

```
GREEDY◹(X[1..m]):
  Y ← ∅
  for i ← 1 to m
      add (X[i], i) to Y
      SATISFY◹(Y, X[i], i)
  return Y
```

```
SATISFY◿(Y, X[i], i):
  maxj ← 0
  for x ← X[i] − 1 down to 1
      if there is a point (x, j) ∈ Y with j > maxj
          maxj ← max{j | (x, j) ∈ Y}
          add (x, i) to Y
```

```
SATISFY◹(Y, X[i], i):
  maxj ← 0
  for x ← X[i] + 1 to n
      if there is a point (x, j) ∈ Y with j > maxj
          maxj ← max{j | (x, j) ∈ Y}
          add (x, i) to Y
```

For each of the following input arrays, give the best estimate you can on the size of the output from all three algorithms. For random inputs, estimate the *expected* output sizes, assuming all $X[i]$ are mutually independent.

(a) $X[i] = i \bmod n$ for all i.

(b) $X[i] = \begin{cases} i \bmod n & \text{for all even } i \\ (n-i) \bmod n & \text{for all odd } i \end{cases}$

(c) $X[i] = iF_{k-1} \bmod F_k$ for all i, where F_k is the kth Fibonacci number and $n = F_k$. (See next page.)

(d) Uniform: $\Pr[X[i] = j] = 1/n$.

(e) Random walk: $\Pr[X[i] = j] = \begin{cases} 1/2 & \text{if } j = (X[i-1] \pm 1) \bmod n \\ 0 & \text{otherwise} \end{cases}$

*(f) Binomial: $\Pr[X[i] = j] = \binom{n}{j}/2^n$

*(g) Zipf: $\Pr[X[i] = j] = 1/jH_n$.

*(h) Zipfibonacci (Fibonaccipf?): $\Pr[X[i] = jF_{k-1} \bmod F_k] = 1/jH_n$, where $n = F_k$.

★(i) Other interesting nontrivial examples.