1. This question asks you to develop a data structure that maintains sequences of numbers, all initially equal to zero, subject to the following operations. (I'll use array *notation* to describe the underlying sequence, but your actual data structure should *not* be an array.)

   - $S \leftarrow \textsc{Init}(n)$: Initialize a new sequence $S[1..n]$ containing n zeros.
   - $\textsc{Shift}(S, i, j, \Delta)$: Add Δ to every number in the interval $S[i..j]$. The number Δ is *not* necessarily an integer; moreover, Δ could be positive, negative, or zero.
   - $\textsc{Scale}(S, i, j, \alpha)$: Multiply every number in the interval $S[i..j]$ by α. The number α is *not* necessarily an integer; moreover, α could be positive, negative, or zero.
   - $x \leftarrow \textsc{Minimum}(S, i, j)$: Return the smallest number in the interval $S[i..j]$.

   Designing this data structure all at once in a single week is a bit much to ask in a single homework. So I'm breaking the design up into steps, extending the deadline by a week, and doubling the credit for this homework. To simplify grading, please start your solution to each part at the top of a new page.

   (a) **[4 points]** Describe a static data structure that supports MINIMUM in $O(\log n)$ *worst-case* time, where n is the length of the stored sequence. Your data structure does not need to support SHFT or SCALE at all.

   (b) **[4 points]** Describe a modification of your data structure from part (a) that supports both MINIMUM and SHIFT, each in $O(\log n)$ *worst-case* time. Your data structure does not need to support SCALE at all. Don't describe the data structure from scratch; instead, describe your changes from part (a), including any necessary changes to your MINIMUM algorithm.

   (c) **[2 points]** Further modify your data structure to support SHIFT, SCALE, and MINIMUM, each in $O(\log n)$ *worst-case* time. Again, only describe your changes from part (b), including any necessary changes to your earlier algorithms. *[Hint: Remember that the scale factor α can be negative.]*

   (d) **[2 points]** Finally, modify your data structure again so that INIT runs in $\boldsymbol{O(1)}$ *worst-case* time; in particular, your INIT does *not* have time to allocate an array of n zeros. Again, only describe your changes from part (c), including any necessary changes to your earlier algorithms.

   For simplicity, you can assume that n is a power of 2, and that arithmetic operations (addition and multiplication) can be performed in $O(1)$ time each.

   [Hint: Use a balanced binary tree with extra information at the vertices. Be lazy.]

Please read the note about partial credit on the next page.

A note about partial credit:

Whenever you are asked to design and analyze an algorithm or a data structure, it is important to keep your priorities straight:

Clarity is more important than correctness.
Correctness is more important than speed.

An algorithm that correctly solves the stated problem is ***always*** better than an algorithm that does not correctly solve the stated problem, even if the correct algorithm runs in *exponential* time. If your algorithm is incorrect, it doesn't matter how fast it is. We provide target running times in part to let you know that those running times are *possible*, but your first goal should be to make something that works at all. Designing a slower solution is often the best first step toward finding a faster one.

Similarly, a clearly-presented result that contains errors is ***always*** better than a correct but badly presented result. If your presentation is poorly written, it doesn't matter whether the algorithm / data structure you're describing is correct. Clearly expressing an incorrect solution is often the best first step toward finding a correct solution, in part because *expressing* the incorrect solution clearly make you *think* about it more clearly, which makes the errors easier to spot.

In particular, we *strongly* recommend writing your algorithms using *pseudocode*—not raw English prose, and *definitely* not compilable C++ or Java or Python or whatever. Structure your pseudocode using standard iterative programming idioms, like indentation, for-loops, while-loops, if-then-else blocks, and function calls. Start each step of your pseudocode on a new line.

Your partial credit for homework reflects these priorities: Slow correct algorithms are worth significantly more partial credit than fast incorrect algorithms; and unreadable solutions are worth nothing, even if they are correct. The same will be true in CS 374 and other later theory courses.

The remaining problems are for you play with on your own.
Discussion in office hours or on Discord is welcome, but don't submit solutions!

---

2. A *maxiphobic heap* is a mergeable priority queue similar to a leftist heap. A maxiphobic heap is a (not necessarily balanced) binary tree, where each node v stores four values:

   - A pointer $v.left$ to the left child of v
   - A pointer $v.right$ to the right child of v
   - A priority $v.priority$, which is larger than the priority of v's parent (if any)
   - An integer $v.size$ equal to the number of descendants of v (including v itself)

   MERGE is implemented as follows:

```
MERGE(u, v):
  if u = NULL
      return v
  if v = NULL
      return u
  if u.priority > v.priority
      swap u ↔ v
  w ← u.left
  x ← u.right
  biggest = max{v.size, w.size, x.size}
  if biggest = v.size
      u.left ← MERGE(w, x)
      u.right ← v
  else if biggest = w.size
      ⟨⟨u.left = w⟩⟩
      u.right ← MERGE(v, x)
  else ⟨⟨biggest = x.size⟩⟩
      u.left ← MERGE(v, w)
      ⟨⟨u.right = w⟩⟩
  return u
```

   Prove that MERGE runs in $O(\log n + \log m)$ time in the worst case, where n and m are the sizes of the two input heaps.

3. Describe and analyze a mergeable priority queue that supports the following operation, in addition to the standard MERGE, INSERT, EXTRACTMIN, and DECREASEKEY:

   - SHIFT(PQ, Δ): Add Δ to the priority of every item in the priority queue PQ.

   MERGE should run in $O(\log n + \log m)$ time, where n and m are the sizes of the two priority queues. INSERT, EXTRACTMIN, DECREASEKEY, and SHIFT should each run in $O(\log n)$ time, where n is the size of the priority queue.

   [Hint: If we didn't have to support MERGE, then supporting SHIFT in $O(1)$ time would be trivial. (Do you see why?) Think about how you would MERGE two different heaps that have had different Δs added to them.]