(25 pts) 1. **(Cyclic Splaying)** Starting from a tree $T_0$ of $n$ nodes a sequence of $\ell \geq 1$ splay operations is done. It turns out that the initial tree $T_0$ and the final tree $T_\ell$ are the same. Let $k$ be the number of distinct nodes splayed in this sequence. (Clearly $k \leq \ell$.) Below is an example where $k = \ell = 4$ and $n = 6$.

(a) Use some setting of node weights to show that the average number of splaying steps in this cycle (i.e., the average per splay operation) is at most $c_1 + c_2 \log_2 n$. (Smaller $c_1$ and $c_2$ is better.)

(b) Now use a different setting of node weights to show that the average number of splaying steps in this cycle (per splay operation) is at most $c_1 + c_2 \log_2 k$. (Smaller $c_1$ and $c_2$ is better.)

**Solution:** Note that since the initial and final trees are the same, the potentials must be equal. By the Access Lemma, as long as all our weights are positive, the average number of splaying steps is $\leq 3(r(t) - r(x)) + 1$.

**a.** Set all the weights of each node to 1. We know $r(t)$, or the rank of the root, is $\lceil \log(s(t)) \rceil$, where $s(t) = \sum_{y \in T(t)} w(y)$. Since all $n$ nodes are in the subtree rooted at the root, $s(t) = n$ and $r(t) = \lceil \log(n) \rceil$. We also know $r(x) \geq 0$, so $3(r(t) - r(x)) + 1 \leq 3[\log(n)] + 1$, so $c_1 = 1$ and $c_2 = 3$.

**b.** Let the weights of the $k$ elements we access be $1$ and the others be $\epsilon > 0$. The rank of the root is then $\lceil \log(k + (n - k)\epsilon) \rceil$. If we choose $\epsilon$ to be small enough so that $(n - k)\epsilon$ is smaller than 1, then $\lceil \log(k + (n - k)\epsilon) \rceil = \lceil \log(k) \rceil \leq \log(k)$. This is because the function $f(x) = \lceil \log(x) \rceil$ only changes when $x$ crosses an integer value. Again by the Access Lemma, we know the average number of splaying steps is $\leq 3(\log(k) - r(x)) + 1 \leq 3 \log(k) + 1$, so $c_1 = 1$ and $c_2 = 3$.

(25 pts) 2. **(Looking for Sum One?)** In the Sum-Query problem, you are given two sorted arrays $A$ and $B$. Each array has $n$ distinct real numbers. You are also given a real number $q$, and you want to find some indices $i, j \in \{1, 2, \ldots, n\}$ such that $A[i] + B[j] = q$, or else report that no such $i, j$ exist.¹ You are only allowed to use the operation $\text{test}(1, j)$ that reports back the result of comparing $q$ with $A[i] + B[j]$.

¹If many different $(A[i], B[j])$ pairs sum to $q$, you are allowed to report any one.
(a) Give an algorithm solving SUM-QUERY using $O(n)$ calls to test(.,.).

(b) In this part we will show that any deterministic algorithm for SUM-QUERY must make at least $n$ calls.  

i. Given some unsorted array $C$ of $n$ real numbers (say in the range $[0, M]$) and a real value $q \in [0, M]$, show that the problem of finding some index $k$ such that $C[k] = q$ (or reporting that no such $k$ exists) requires $n$ comparisons between $q$ and elements of $C$ in the worst case.

ii. Given such an array $C$ of $n$ real numbers and a real value $q$, all in the range $[0, M]$, show how to construct two sorted arrays $A$ and $B$ such that

- $C[i] := A[i] + B[n + 1 - i]$, and
- if $i + j < n + 1$, then $A[i] + B[j] < 0$, and
- if $i + j > n + 1$, then $A[i] + B[j] > M$.

This construction should require no comparisons between elements.

iii. Use the above parts to infer a lower bound of $n$ calls to test(.,.) for the SUM-QUERY problem.

Solution: (a) Our algorithm is written in pseudocode below.

```plaintext
(a, b) = (1, n)
while a <= n and b >= 1:
    cmp = test(A[a], B[b])
    if cmp == "less":
        a++
    else if cmp == "greater":
        b--
    else:
        return (a, b) # A[a] + B[b] = q
return (null, null) # no such a, b exist
```

On every iteration of our algorithm, we do exactly one call to $|\text{test}(.,.)|$ and there can be at most $2n - 1$ iterations since we either increment $a$ or decrement $b$, both of which can be done $n - 1$ times, plus the first iteration. Thus, there are $O(n)$ calls to $|\text{test}(.,.)|$.

Finally, we prove the correctness of our algorithm. Say that there do exist some $i, j$ such that $A[i] + B[j] = q$. Our algorithm sweeps across $A$ and $B$ one element at a time, so eventually $a = i$ and $b = j$ (but not necessarily at the same time). If they do happen at the same time, then $|\text{test}(.,.)|$ will return "equal" and we are done. Otherwise, without loss of generality, say $a = i$ before $b = j$. Since $b$ starts off at $n$, $b > j$. $B$ is sorted with all distinct numbers, so $B[b] > B[j]$. We know $A[i] + B[j] = q$, so $A[i] + B[b] > q$, so $|\text{test}(.,.)|$ will return "greater" and we will decrement $b$. Eventually, $b = j$, $|\text{test}(.,.)|$ will return "equal" and our algorithm will terminate correctly. If there are no $i, j$ such that $A[i] + B[j] = q$, our algorithm will return $|(null, null)|$ since $|\text{test}(.,.)|$ will never return "equal".

(b) (i) Consider some instance where $q$ does not belong to $C$. If we do at most $n - 1$ comparisons, there must be some element $C[i]$ we do not compare with $q$. Then simply make $C[i] = q$, our algorithm will still say $q$ was not in $C$, which is incorrect. Thus, any algorithm must look at all $n$ elements.

2Since arrays $A$ and $B$ are sorted, you cannot just claim that “you have to look at all numbers”.


Suppose we have a binary counter such that the cost to increment or decrement the counter is equal to the number of bits that need to be flipped. We saw in class that if the counter begins at 0, and we perform $n$ increments, the amortized cost per increment is just $O(1)$. Equivalently, the total cost to perform all $n$ increments is $O(n)$.

Suppose that we want to be able to both increment and decrement the counter.

(a) Show a sequence of $n$ operations allowing both increments and decrements and starting from 0 that, without ever making the counter go negative, costs $\Omega(\log n)$ amortized per operation (i.e., $\Omega(n \log n)$ total cost).

**Solution:** Let $2^{b+1} \leq n \leq 2^{b+2}$. Use $2^b \leq n/2$ operations to transform the number to a 1 followed by $b$ 0’s. This takes at least $2^b$ work. Then alternately decrement and increment the number for the remaining $\geq 2^b$ steps. Each of these operations takes $b + 1$ work. So, the total work over all these operations is at least $2^b + (b + 1)2^b = (b + 2)2^b$. Thus the amortized cost per operation is more than $b2^b/n \geq b2^b/2^{b+2} = b/4 = \Omega(\log n)$.

To reduce the cost observed in part (a) we’ll consider the following redundant ternary number system. A number is represented by a sequence of trits, each of which is 0, +1,
or $-1$. The value of the number represented by $t_{k-1}, \ldots, t_0$ (where each $t_i, 0 \leq i \leq k-1$ is a trit) is defined to be
\[
\sum_{i=0}^{k-1} t_i 2^i.
\]
For example, \[1 \overline{0} \overline{1}\] is a representation for $2^2 - 2^0 = 3$.

The process of incrementing a ternary number is analogous to that operation on binary numbers. You add 1 to the low order trit. If the result is 2, then it is changed to 0, and a carry is propagated to the next trit. This process is repeated until no carry results. Decrementing a number is similar. You subtract 1 from the low order trit. If it becomes -2 then it is replaced by 0, and a borrow is propagated. Note that the same number may have multiple representations (e.g., \[1 \overline{0} \overline{1}\] = \[1 \overline{1} \overline{1}\]). That’s why this is called a redundant ternary number system. The cost of an increment or a decrement is the number of trits that change in the process.

(b) Starting from 0, a sequence of $n$ increments and decrements is done. Give a clear, coherent proof that with this representation, the amortized cost per operation is $O(1)$ (i.e., the total cost for the $n$ operations is $O(n)$). Hint: think about a “bank account” or “potential function” argument.

Solution: When the number is incremented or decremented, some 1s may change to 0 and some $-1$s may change to 0, but a 1 will never directly change to $-1$ or vice versa. This means that if $2$ is paid each time a 0 is changed (to 1 or $-1$), there will be enough money in the bank to pay for converting it back to a 0. (Think of having a separate bank account for each digit.) The second fact is that in any increment or decrement, at most one 0 changes (to either 1 or $-1$). Therefore, the amortized cost per operation is $\leq 2$. 