Binary Counter Revisited: Suppose we are incrementing a binary counter, but instead of each bit flip costing 1, suppose flipping the $i^{th}$ bit costs us $2^i$. (Flipping the lowest order bit $A[0]$ costs $2^0 = 1$, the next higher order bit $A[1]$ costs $2^1 = 2$, the next costs $2^2 = 4$, etc.) What is the amortized cost per operation for a sequence of $n$ increments, starting from zero?

Solution: $O(\log n)$. The idea is simple. We flip $A[0]$ each time, so pay $n$ over $n$ operations. We flip $A[1]$ every other time, so pay $\leq 2 \times n / 2 = n$ over $n$ operations, and so on, until $A[\lfloor \log_2 n \rfloor]$ which gets flipped once for a cost of at most $n$. Hence $O(n \log n)$ in total, or $O(\log n)$ per operation.

Binary Counter Re-Revisited: Suppose we are incrementing a binary counter, but instead of each bit flip costing 1, suppose flipping the $i^{th}$ bit costs us $i + 1$. (Flipping the lowest order bit $A[0]$ costs $0 + 1 = 1$, the next higher order bit $A[1]$ costs $1 + 1 = 2$, the next costs $2 + 1 = 3$, etc.) What is the amortized cost per operation for a sequence of $n$ increments, starting from zero?

Another Dictionary Data Structure: A “dictionary” data structure supports fast insert and lookup operations into a set of items. Yesterday you saw splay trees where both inserts and lookups can be done with amortized cost only $O(\log n)$ each. Note that a sorted array is good for lookups (binary search takes time only $O(\log n)$) but bad for inserts (takes linear time), and a linked list is good for inserts (takes constant time) but bad for lookups (takes linear time). Here is a simple method that takes $O(\log^2 n)$ search time and $O(\log n)$ amortized cost per insert.
Here, we keep a collection of arrays, where array \( i \) has size \( 2^i \). Each array is either empty or full, and each is in sorted order. However, there will be no relationship between the items in different arrays. The issue of which arrays are full and which are empty is based on the binary representation of the number of items we are storing. For example, if we had 11 items (where \( 11 = 1 + 2 + 8 \)), then the arrays of size 1, 2, and 8 would be full and the rest empty, and the data structure might look like this:

\[
\begin{align*}
A0 & : [5] \\
A1 & : [4,8] \\
A2 & : empty \\
A3 & : [2, 6, 9, 12, 13, 16, 20, 25]
\end{align*}
\]

**Lookups.** How would you do a lookup in \( O(\log^2 n) \) worst-case time?

**Inserts.** How would you do inserts? Suppose you wanted to insert an element, you will have 12 items and \( 12 = 8 + 4 \), you want to have two full arrays in \( A2 \) and \( A3 \) and the rest empty. Suggest a way that, if you insert an element 11 into the example above, gives:

\[
\begin{align*}
A0 & : empty \\
A1 & : empty \\
A2 & : [4, 5, 8, 11] \\
A3 & : [2, 6, 9, 12, 13, 16, 20, 25]
\end{align*}
\]

(Hint: merge arrays!)

**Cost of Lookups:** Suppose the cost of creating an array of length 1 costs 1, and merging two arrays of length \( m \) costs \( 2m \). So, the above insert had cost \( 1 + 2 + 4 \). Inserting another element would cost 1, and the next insert would cost \( 1 + 2 \).

What is the amortized cost of \( n \) inserts?
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 $1 + 3 \log_2 n$. Make use of the Access Lemma for splay trees covered in lecture yesterday.

(b) (Extra material, for you to do at home.) 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 also most $1 + 3 \log_2 k$. 

\[ \begin{array}{cccccc}
4 & 3 & 0 & 5 & 4 \\
/ \ 2 \ 5 \ splay(3) / \ 2 \ 4 \ splay(0) / \ 2 \ splay(5) / \ 2 \ splay(4) / \ 2 \ 5 \\
/ \ 0 \ 3 \ splay(3) / \ 0 \ 3 \\
/ \ \ splay(5) / \ splay(5) / \ splay(4) / \\
/ \ \ \ \ splay(5) / \\
0 \ 3 \\
\end{array} \]