Competitive Algorithms: Searching for Keys
You live at position 0 of the PA turnpike. You've dropped your keys somewhere (at an integer location) and need to find them. The keys are small, so you cannot see them from afar, you will only see them when you are at the same location as them. If the keys are at location \( X \in \mathbb{R} \), then the optimal solution is just to go to that location, and travel \(|X|\). If you knew the sign of \( X \), you could do that. But you don’t.

What should you do to minimize the distance traveled? Your algorithm should have constant competitive ratio, ideally 9.

**Solution:** Do “doubling search”. Go to positions \( +1, -2, +4 = 2^2, -8 = -2^3, +16 = 2^4 \), etc, until you find the keys. Suppose \( X \) lies in \((2^{2i}, 2^{2i+2}]\). Then the total distance you travel is

\[
2(1 + 2 + 4 + \ldots + 2^{2i} + 2^{2i+1}) + X = 2 \cdot (2^{2i+2} - 1) + X \leq 9X.
\]

For negative \( X \), the calculation is similar. Indeed, if \( X \in [-2^{2i+1}, -2^{2i-1})\). Then the total distance you travel is

\[
2(1 + 2 + 4 + \ldots + 2^{2i-1} + 2^{2i}) + |X| = 2 \cdot (2^{2i+1} - 1) + |X| \leq 9|X|.
\]

Competitive Algorithms: Examples for Non-MTF Algorithms
During lecture, we covered several solutions to the List Update problem with competitive ratio \( \Omega(n) \)

- **Do Nothing:** Don’t reorder the list.

- **Single Exchange:** After accessing \( x \), if \( x \) is not at the front of the list, swap it with its neighbor toward the front.

Now we will be covering an alternative algorithm and proving the same competitive ratio

- **Frequency Count:** Maintain a frequency of access for each item. Keep the list ordered by non-increasing frequency from front to back.

Give request sequences for the List Update problem where the following algorithms have competitive ratio \( \Omega(n) \).

**Solution:** Say the starting list is \( 1, 2, \ldots, n \). Then

1. \( n, n, \ldots \) If the number of requests \( T \) is very large, opt moved \( n \) to the front and \( n + T \) for \( T \) requests, whereas we pay \( nT \). So the competitive ratio is \( \frac{nT}{n+T} \), which is \( \Omega(n) \) when \( T \geq n \).
2. $n, n-1, n, n-1, \ldots$. OPT will move both to the front, and pay $O(n)$ for this, and $O(1)$ per request after that. We will pay $\Omega(n)$ each time.

3. $1^T, 2^T, 3^T \ldots n^T$. The list does not change, so we will pay $T + 2T + \ldots + nT = \Theta(n^2T)$. The optimal solution is to move each element to the front the first time we access it, so we pay $\Theta(n^2 + nT)$ in total. Hence the competitive ratio is $\Omega(n)$ when $T \geq n$. 
Counting with suffix trees

Given a string $S$ of length $n$, for each $1 \leq i \leq n$, we want to find the number of strings that occur as substrings of $S$ exactly $i$ times. How fast can you do this? (Hint: suffix trees)

Solution:

Create a suffix tree for the string $S$. For each node in the suffix tree, compute a "leaf count" field, which is the number of leaves in the subtree rooted there. It is easy to compute this using a DFS of the tree. Also for any node $x$ in the suffix tree, let $p(x)$ denote the parent of $x$. Let $LC(x)$ denote the leaf count of $x$.

Suppose that $x$'s leaf count is $i$. This means that the string represented by $x$ occurs exactly $i$ times in the tree. The same is true for every string that traverses down from the root and stops somewhere below $p(x)$ but above or at $x$ i.e. along the edge from $p(x)$ to $x$. The number of such strings is the length of the edge from $p(x)$ to $x$. Denote the length of this edge by $L(x)$.

Now to solve the problem create an array $A[]$ initialized to zero. $A[i]$ will count the number of substrings that occur exactly $i$ times. So we iterate through all non-root nodes $x$ of the suffix tree, and do the following assignment:

$$A[LC(x)] \leftarrow A[LC(x)] + L(x)$$

The array $A$ now contains our answer. The running time is $O(n)$. 