1. For adaptive prefix-free coding, what is the property that allows Alice to change the code without confusing Bob?

**SOLUTION:** The code is changed in a deterministic way that depends only on the already sent characters.

2. Suppose you have want to maintain a list of $\sigma$ elements under the following operations:
   - given an element, return its current position in the list and move it to the front,
   - given a position, return the element currently in that position and move it to the front.

Design a data structure for this problem with the best time- and space-complexities you can achieve (with a reasonable effort).

**SOLUTION:** One example of an easy solution to get $O(\log \sigma)$ time is use e.g. AVL-tree implementation of a priority queue, and an array. The array maps the characters to their timestamps (of the most recent occurrence) and AVL-tree holds the timestamps, with pointers to the characters. If you use timestamps instead of indexes, you won’t have to change any of the other entries when moving to front.

When given an element, check the array for its timestamp (constant time for hash) and return the timestamp. Delete that node in the tree in $O(\log \sigma)$ time, and insert a new one with new timestamp in $O(\log \sigma)$ time, it will go to front because it’s a priority queue. Then update the array in constant time.

When given a position, find it in the tree in $O(\log \sigma)$ time and check the pointer for the element. Return the element. Delete the node in $O(\log \sigma)$ time and insert new one in $O(\log \sigma)$ time. Update the array.

To use time proportional to the total length of the encoding, instead of starting searching at the root of the AVL tree, we can start searching at its leftmost node. We walk up the AVL tree from the leftmost node toward the root until we find a timestamp that’s bigger than our target timestamp, then search in the bigger timestamp’s subtree. For more details, see [http://dl.acm.org/citation.cfm?id=5688](http://dl.acm.org/citation.cfm?id=5688).

3. Explain how, given $c$, we can modify MTF so that it uses roughly $\sigma^{1/c}$ space while increasing the encoding length by a factor of at most $c$.

**SOLUTION:** As a reminder, the space requirement of MTF algorithm is set by the length of the list that stores the order in which the characters have last appeared. When the whole alphabet is stored on that list, the space requirement is $O(\sigma)$.

So, to use less space, shorten the list to length $\sigma^{1/c}$. Then if something is on the list, write its index $i$ on the list using $\log i$ bits (using e.g. Elias $\gamma$), and if it’s not on the list, use $\log \sigma$ bits to represent it (e.g. fixed-length binary encoding). (Then move it to the front of the list as in normal MTF, if this character was not already on the list, the last character on the list drops off.)

How to prove that this increases encoding length by a factor of at most $c$: If the character is not on the list, the number of distinct characters since its last occurrence must have been at least $\sigma^{1/c} + 1$ (that is, on the original list its smallest possible index would have
been $\sigma^{1/c} + 1)$. So if we encoded the character using its index in the original list, it would use $\log(\sigma^{1/c} + 1) = \frac{1}{c} \log \sigma$ bits. 

As a practical matter, you will need an extra bit to tell Bob whether you’re sending an index or a character, that he knows how to decode it.

4. The slides on arithmetic coding mistakenly say we can map the interval $[0.010000001, 0.0100011)$ to $[0, 1)$. What should the second interval be and why?

SOLUTION: As Alice is sending the first 5 bits after binary point (the shared prefix of the interval boundaries), they, and only they, are discarded. Therefore the second interval should be $[0.001, 0.11)$. Alice cannot discard any more bits, because Bob doesn’t know those bits!