1. **Kleinberg & Tardos Chapter 13, question 8**

2. Show how to construct a dictionary data structure that, with probability $1 - \delta$, supports lookup and delete in $O(1)$ time and inserts in $O(1)$ amortized time per insert when the number of inserts $n$ is not known in advance. Your data structure should only require an amount of memory bounded by a polynomial (in $n$ and $1/\delta$).

For this problem, you may assume that you have a function that on input $k$ returns a $k$-bit prime number (i.e., between $2^k$ and $2^{k+1}$) and runs in time polynomial in $k$. You may also assume that allocating any amount of memory initialized to NULL can be done in $O(1)$ time. (Note: this last assumption can be removed, but you are not required to address this.)

3. A **self-organizing data structure** is reorganized during execution in response to a sequence of operations, with the goal of achieving good performance on any given sequence. Often, the majority of operations concern a small number of elements, and so optimizing the access time to these elements can improve performance overall. In this problem, we will analyze a simple self-organizing linked list. In this linked list, whenever we access a list node, we move that node to the head of the list (pushing the rest of the elements back one position). We’ll say that it costs one operation ($1$, if you like) to access a node of the list and check its value, so that walking a list to access the $k$th element of that list costs $k$ operations.

   (a) Consider some arbitrary initial list containing $n$ distinct elements, and suppose we have used the self-organizing algorithm to access some $k$ out of the $n$ elements. For each of the remaining $n - k$ elements, exactly how many more operations does it take to access that element in the self-organizing list than in the original, fixed list? (Note that this may depend on the ordering of the original list and which $k$ elements were accessed! You should identify and describe what the relevant properties of the ordering are.)

   (b) Now, show that for any fixed list containing $n$ distinct elements (for example, just the list where the elements are sorted), and any sequence of $T$ accesses to the list elements, the self-organizing algorithm applied to this list uses at most twice as many operations as if we had not rearranged the list.

   (c) Finally, we will consider the case where the self-organizing list is executed on a different initial ordering than some arbitrary static list. Show that for every possible ordering of the $n$ distinct elements and sequence of $T$ accesses, if these accesses to that static ordering cost $C$ operations in total, then the self-organizing algorithm started from an
arbitrary, fixed initial ordering (again, for example with the elements sorted) uses at
most $2C + n^2$ operations in total, for a competitive ratio of at most $2 + \frac{n^2}{T} \xrightarrow{T \to \infty} 2$
against all static lists.