Reminder: You may work in groups and use outside sources. But, you must write up solutions in your own words and properly reference your sources for each problem. This includes listing your collaborators and properly citing any sources you use. Solutions to each problem must be electronically typeset in separate documents and submitted online via Canvas.

Problem 1-1. Kleinberg & Tardos, Chapter 4, Problem 4.

Here, I will give you the algorithm and your obligation is to prove that it is correct. Lets say you have a sequence $S = s_1, s_2, ..., s_n$ that consists of events and sequence $S' = s'_1, s'_2, ..., s'_m$ and you want to check if $S'$ is a subsequence of $S$. We start with $s'_1$ and walk down $S$ until we find a match, say at position $k_1$ — that is $s_{k_1} = s'_1$ and this is the first instance this is true. We then start at position $k_1 + 1$ and keep walking down $S$ until we find $s'_2$, say at position $k_2$ and so on. If we find all of $S'$ — that is, we find $k_1, k_2, ..., k_m$ such that $s'_i = s_{k_i}$ for all $1 \leq i \leq m$, then we declare success and return $k_1, k_2, ..., k_m$. If we don’t find all matches before $S$ ends, we declare failure.

(a) Write the pseudocode for the algorithm I have described here. Analyze the running time of this algorithm (asymptotically).

(b) It should be pretty clear that if the algorithm finds a match — that is, it finds $k_1, k_2, ..., k_m$, then it is a valid solution. The difficulty is in proving that if the algorithm returns failure, then no solution exists and $S'$ is not a subsequence of $S$. In other words, you want to prove that if a solution exists, then the greedy algorithm always finds it. The proof is similar to the one we say in class for the interval scheduling problem; that is, we will prove that the greedy algorithm stays ahead.

Hint: Assume that there is a solution — $s_{l_1}, s_{l_2}, ..., s_{l_m}$ matches $S'$ such that $l_j < l_{j+1}$ for all $j$. Think of a stay-ahead property in terms of these $l_1, ..., l_m$’s and $k_1, k_2, ...$ that our algorithm finds. Can you think of a stay ahead property that you can prove by induction?

Problem 1-2. Dimensionality reduction

Dimensionality reduction is a widely-used technique for accelerating computations on high-dimensional data, by replacing them with a lower-dimensional “summary” that preserves the key properties of the original data, typically the approximate distances of points. The key step in fast dimensionality reduction is multiplying a vector by a matrix from the following family of special matrices, $H_0, H_1, \ldots, H_k, \ldots:$
• $H_0$ is the $1 \times 1$ matrix $[1]$.

• For $k > 0$, $H_k$ is the $2^k \times 2^k$ matrix

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

(a) If you naively multiply a length $n$ vector with a length $n \times n$ matrix, what is the running time?

(b) What makes the method described above so fast is that, for a vector $\vec{v}$ of length $n = 2^k$, the matrix-vector product $H_k \vec{v}$ can be computed in time $O(n \log n)$. Give a recursive, divide and conquer algorithm for this. Show how to do this (and as usual, prove the correctness and time bound of your algorithm).

*Hint:* The idea is similar to matrix multiplication. In particular, given a problem of size $n$ (where you multiply a vector of size $2^k$ with a matrix $H_k$ of size $2^k \times 2^k$) can you find the solution two subproblems of size $n/2$? How much can you afford for the divide/combine steps of the divide and conquer algorithm?

**Problem 1-3. The Stock Market Problem**

The problem with the stock market is that, while it is possible to make a great deal of money buying and selling stocks, it’s easy to lose even more. The long-standing—if somewhat unhelpful—maxim to make more money than you lose is “buy low, sell high.”

We are going to solve the posthoc stock market problem. The stock market problem is finding the best opportunity to follow this advice: for any sequence of integer prices, where the index in the sequence represents time, find maximum jump from an earlier price to a later price. For example, if the sequence of prices was

\[(40, 20, 0, 0, 0, 1, 3, 3, 0, 0, 9, 21)\]

then the maximum jump is 21, which happens between the price at time 2 and time 11. More formally, the stock market problem is to compute

$$\max \{s_j - s_i | 1 \leq i \leq j \leq |s|\}$$

Here, we are going solve the stock market problem. Solve the stock market problem by divide-and-conquer recursive programming, argue its correctness, and analyze its asymptotic running time.