1. Cholesky Decomposition.

(a) Write a function with argument \( n \) to generate a random symmetric \( n \times n \)-positive definite matrix. To do this:
- generate an \( n \times n \) matrix \( C \) whose entries are independent normal random variables;
- return \( CC^T \).

Check your matrices are positive definite using the \texttt{eigen()} function.

(b) Implement the recursive Cholesky decomposition algorithm from the lecture.

(c) Test it using your function for generating positive definite matrices, and by comparing the answers to \texttt{chol()}.

(d) Create a function which takes a vector \( \mu \) and a symmetric positive definite matrix \( \Sigma \) and uses them to generate a multivariate normal vector \( N_n(\mu, \Sigma) \). Your function should check that \( \Sigma \) is positive definite using \texttt{eigen()} and symmetric using \texttt{isSymmetric()}.

2. Sorting. Here is an algorithm called ‘Quicksort’ for sorting the objects in a vector.

Function: sort a vector \( x \)
Input: vector \( x \) of length \( n \)
Output: a vector \( Q(x) \) containing entries of \( x \) arranged in ascending order

1. if \( n \leq 1 \) return \( x \);
2. pick an arbitrary ‘pivot’ element \( i \leq n \);
3. let \( z = (x_j \mid x_j < x_i) \) and \( y = (x_j \mid x_j > x_i) \);
4. let \( z' = Q(z) \) and \( y' = Q(y) \); [\textit{i.e. call the algorithm on the smaller vectors}]
5. let \( x' \) be the entries in \( x \) not used in \( y \) or \( z \); [\textit{i.e. any entries equal to} \( x_i \)]
6. return \( (z', x', y') \).

(a) Implement the algorithm in R, and test it on some random numbers.

(b) What is the complexity if \( x_i \) is always the smallest element?

(c) Show that, if the pivot \( x_i \) is the median element on each call, that the complexity is at most \( O(n \log_2(n)) \).
3. **Back Solving.** Here is a recursive algorithm to solve $Ax = b$ where $A$ is an upper triangular matrix, using back substitution.

Function: solve $Ax = b$ for $x$ by back-substitution

Input: $n \times n$ upper triangular matrix $A$ and vector $b$ of length $n$

Output: vector $x$ of length $n$ solving $Ax = b$

1. If $n = 1$ return $x = b/A$;
2. create a vector $x$ of length $n$;
3. set $x_n = b_n/A_{nn}$;
4. set $b' = b_{1:(n-1)} - A_{[1:(n-1), n]}x_n$;
5. set $A' = A_{[1:(n-1), 1:(n-1)]}$;
6. solve $A'x' = b'$ for $x'$ by back-substitution ;
7. set $x_{1:(n-1)} = x'$;
8. return $x$.

(a) Implement this algorithm as a recursive function in R. Your function should take as input an upper triangular $n \times n$ matrix $A$ and return a solution $x$ satisfying $Ax = b$.

(b) For $n = 10$, create an $n \times n$ upper triangular matrix $A$ and a vector $b$ of length $n$. Check the solution from your function against `backsolve()` and `solve()`.

4. **Longest Increasing Subsequence.**

The object of this exercise is to write a function that, given a sequence of numbers $a = (a_1, \ldots, a_k)$, returns $Q(a) = (a_{s_1}, \ldots, a_{s_L})$, the longest subsequence of $a$ such that $a_{s_1} < \cdots < a_{s_L}$. [Note that it is implicit in the idea of a subsequence that $s_1 < \cdots < s_k$.

(a) Write a function that, for each $i$, recursively calculates the longest increasing subsequence of $(a_1, \ldots, a_{i-1}, a_i)$ that ends with $a_i$. [Hint: remove the final element of $a$ and invoke the function on this shorter vector; then add $a_k$ to the longest subsequence whose final element is less than $a_k$.

(b) Use this to return a function that solves the problem of finding $Q(a)$.

(c) Calculate the computational complexity of this method.