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 `eigen()` function.

```r
randPDmat <- function(n) {
  C <- matrix(rnorm(n^2), n, n)
  out <- C %*% t(C)
  out
}
```

```r
A <- randPDmat(8)
all(eigen(A)$value > 0) # are all eigenvalues positive?
[1] TRUE
```

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

```r
# Function to find Cholesky decomposition of symmetric
myChol <- function(A) {
  n <- dim(A)[1]
  if (dim(A)[2] != n)
    stop("A must be a square matrix") # check n x n
  if (n == 1)
    return(sqrt(A))
  L <- matrix(0, n, n)
  L[1, 1] <- sqrt(A[1, 1]) #count as 1 op
  L[1, 2:n] <- rep(0, n - 1)
  A22 = A[2:n, 2:n, drop = FALSE]
  newA = A22 - L[2:n, 1] %*% t(L[2:n, 1]) #2(n-1)^2 here
  # but n(n-1) possible
  L[2:n, 2:n] = myChol(newA)
  return(L)
}
```

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

```r
A <- randPDmat(6)
## chol() gives an upper triangular matrix,
## but can compare transpose to our answer:
t(chol(A)) - myChol(A) # all numerically 0
```
(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 `eigen()` and symmetric using `isSymmetric()`.

```r
mvnorm <- function(mu, Sigma) {
  n <- nrow(Sigma)
  ## check matrix is valid
  if (min(eigen(Sigma)$values) <= 0)
    stop("Sigma must be positive definite")
  if (!isSymmetric(Sigma))
    stop("Sigma must be symmetric")
  ## use method from lecture
  L <- myChol(Sigma)
  z <- rnorm(n)
  x <- mu + L %*% z
  return(c(x))
}
```

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 \( z = (x_j | x_j < x_i) \) and \( y = (x_j | x_j > x_i) \);
4. let \( z' = Q(z) \) and \( y' = Q(y) \); [i.e. call the algorithm on the smaller vectors]
5. let \( x' \) be the entries in \( x \) not used in \( y \) or \( z \); [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.

```r
quickSort <- function(x) {
  n <- length(x)
  if (n <= 1)
    return(x)
  i <- sample(n, 1) # pick a pivot at random
  z <- x[x < x[i]]
  y <- x[x > x[i]]
  return(c(quickSort(z), x[i], quickSort(y)))
}
```
xis <- x[x == x[i]]  # in case of ties

return(c(Recall(z), xis, Recall(y)))
}

x <- rnorm(10000)
out <- quickSort(x)

(b) What is the complexity if \( x_i \) is always the smallest element?
In this case we see \( g(n) = 2n + g(n-1) + g(0) \), so \( g(n) = O(n^2) \). The algorithm relies on being able to divide the problem up to be efficient, so picking the smallest element doesn’t work very well.

(c) Show that, if the pivot \( x_i \) is the median element on each call, that the complexity is at most \( O(n \log_2 n) \).
In this case we get a recursion of the form \( g(n) = 2n + 2g((n - 1)/2) \). Now suppose that \( g(k) \leq Mk \log_2 k \) for some \( M \) and all \( k < n \). Then

\[
g(n) \leq 2n + 2M \frac{n}{2} \log_2 \frac{n}{2}
= 2n + Mn \log_2 n - Mn
\leq Mn \log_2 n
\]

provided that \( M \geq 2 \). Hence \( g(n) = O(n \log_2 n) \).

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 \( Ax = b \).

```r
backSolve <- function(A, b) {
  n <- length(b)
  if (nrow(A) != ncol(A))
    stop("A must be a square matrix")
  if (nrow(A) != n)
    stop("Dimensions of A and b must match")
```
\( x = b[n]/A[n, n] \)
\[
\text{if } (n == 1) \\
\quad \text{return}(x) \\
A2 \leftarrow A[-n, -n, \text{drop} = \text{FALSE}] \\
b2 \leftarrow b[-n] - A[-n, n] * x \\
\]
\( x = \text{c(backSolve}(A2, b2), x) \)
\[
\text{return}(x) 
\}

(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 \text{backsolve()} and \text{solve()}.

\[
\text{A} \leftarrow \text{matrix}(0, 10, 10) \\
\text{A}[-\text{upper.tri}(A, \text{diag} = \text{TRUE})] = \text{rnorm}(55) \\
\text{b} \leftarrow \text{rnorm}(10) \\
\text{backSolve}(A, b) \\
\]  
\[
\begin{bmatrix}
-1.80e+08 & -2.92e+07 & 5.33e+06 & 1.28e+06 & -1.76e+04 & 1.59e+04 & -7.46e+02 \\
-1.52e+02 & 4.66e+01 & 1.52e-01 \\
\end{bmatrix}
\]
\[
\text{solve}(A, b) \\
\]  
\[
\begin{bmatrix}
-1.80e+08 & -2.92e+07 & 5.33e+06 & 1.28e+06 & -1.76e+04 & 1.59e+04 & -7.46e+02 \\
-1.52e+02 & 4.66e+01 & 1.52e-01 \\
\end{bmatrix}
\]

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 \( s_1 < \cdots < s_L \).]

(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 \( a_k \).]

```r
## Return longest increasing subsequences for first i entries input: x - a numeric vector output: a list of the same length as x, whose ith entry is the longest increasing subsequence of x[1],...,x[i] that ends with x[i].
liseqs <- function(x) { 
  ## check length of x and finish if <= 1. 
  n <- length(x) 
  if (n == 0) 
    return(list()) else if (n == 1) 
    return(list(x)) 

  ## remove last element and recall function 
  x_s <- x[-n] 
  tmp <- Recall(x_s)
```
if last element is smallest, longest sequence is just that value
if (min(x) == x[n])
    return(c(tmp, list(x[n])))

## now get lengths of these subsequences
len <- lengths(tmp, FALSE)

## attach x[n] to longest subsequence whose final element is smaller
wh <- which.max(len * (x_s <= x[n]))  # longest we can add x[n] to
out <- c(tmp, list(c(tmp[[wh]], x[n])))
    out
}
liseqs(rnorm(10))

[[1]]
[1] -0.39

[[2]]
[1] -0.390 -0.123

[[3]]
[1] -0.390 -0.123  0.166

[[4]]
[1] -1.51

[[5]]
[1] -0.390 -0.123  0.166  0.221

[[6]]
[1] -0.390 -0.123  0.166  0.221  0.780

[[7]]
[1] -0.390 -0.123  0.166  0.177

[[8]]
[1] -1.509 -0.416

[[9]]
[1] -1.509 -0.416 -0.196

[[10]]
[1] -1.509 -0.416 -0.254

(b) Use this to return a function that solves the problem of finding $Q(a)$.
This is now rather trivial.

longIncSub <- function(x) {
    ## invoke the earlier function, and return the longest subsequence
    tmp <- liseqs(x)
wh <- which.max(lengths(tmp))
return(tmp[[wh]])

longIncSub(rnorm(10))
[1] -1.206 -1.101 -0.375 0.108 3.526

(c) Calculate the computational complexity of this method.

It is not hard to see that the code above just has to search through the list of vectors ending with $a_1, \ldots, a_{k-1}$ to find the longest, so this is an operation that is just linear in $k$. Since this is recursed we have the relation $f(k) = O(k) + f(k - 1)$, and it is easy to check that this implies that $f(k) = O(k^2)$. 