

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.

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

A <- randPDmat(8)
all(eigen(A)$value > 0) # are all eigenvalues positive?

[1] TRUE
```

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

```
# 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[2:n, 1] <- A[2:n, 1]/L[1, 1] #n-1 ops
  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()`.

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

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]
[1,]	0	0	0	0.00e+00	0	0
[2,]	0	0	0	0.00e+00	0	0
[3,]	0	0	0	0.00e+00	0	0
[4,]	0	0	0	2.22e-16	0	0
[5,]	0	0	0	-1.11e-16	0	0
[6,]	0	0	0	-2.78e-17	0	0

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

```
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 $z = (x_j \mid x_j < x_i)$ and $y = (x_j \mid x_j > x_i)$;
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.

```
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]]
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

$$\begin{aligned}
g(n) &\leq 2n + 2M \frac{n}{2} \log_2 \frac{n}{2} \\
&= 2n + Mn \log_2 n - Mn \\
&\leq Mn \log_2 n
\end{aligned}$$

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 $Ax = b$.

```
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]
  if (n == 1)
    return(x)
  A2 <- A[-n, -n, drop = FALSE]
  b2 <- b[-n] - A[-n, n] * x

  x = c(backSolve(A2, b2), x)

  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 `backsolve()` and `solve()`.

```
> A <- matrix(0, 10, 10)
> A[upper.tri(A, diag = TRUE)] = rnorm(55)
> b <- rnorm(10)
> backSolve(A, b)

[1] 977.49 95.75 -526.03 -44.26 17.96 35.75 4.99 1.40
[9] 0.67 1.36

> solve(A, b)

[1] 977.49 95.75 -526.03 -44.26 17.96 35.75 4.99 1.40
[9] 0.67 1.36
```

4. Longest Increasing Subsequence.*

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

- (a) Write a function that, for each i , recursively calculates the longest increasing subsequence of $(a_1, \dots, a_{i-1}, a_i)$ that ends with a_i . [Hint: remove the final element of \mathbf{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 .]

```
## 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] 0.553

[[2]]
[1] 0.553 0.559

[[3]]
[1] -0.124

[[4]]
[1] -1.15
```

```

[[5]]
[1] -0.124  0.384

[[6]]
[1] -1.15 -1.06

[[7]]
[1] 0.553 0.559 0.757

[[8]]
[1] 0.553 0.559 0.757 1.817

[[9]]
[1] 0.553 0.559 0.581

[[10]]
[1] -1.153 -1.060 -0.161

```

- (b) Use this to return a function that solves the problem of finding $Q(\mathbf{a})$.
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] -0.0537  0.1745  0.5944  0.8791

```

- (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, \dots, 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 $f(k) = O(k^2)$.