1. (30) We have n people, from which we need to select k for a committee. Suppose I choose such a committee at random, and you do too. The code below will find the probability that your choices and mine agree in at least one case. Fill in the blanks:

```r
commsim <- function(n,k,nreps) {
  nagree <- 0
  for (___________) {
    comm1 <- choosecomm(n,k)
    comm2 <- choosecomm(n,k)
    if (____________) > 0)
      nagree <- nagree + 1
  }
  return(nagree/nreps)
}
```

```r
choosecomm <- function(n,k) {
  chosen <- vector(length=k)
  remaining <- 1:n
  for (i in 1:k) {
    # choose j at random from 1..length(remaining)
    j <- floor(runif(1) * length(remaining)) + 1
    chosen[i] <- ____________________
    remaining <- ____________________
  }
  return(chosen)
}
```

2. (30) Here you will write an R class “ut” for upper-triangular matrices. Recall that this means that these are square matrices whose elements below the diagonal are 0s. For example:

```
1 5 12
0 6 9
0 0 2
```

The component mat of this class will store the matrix. There is no point in storing the 0s, so only the diagonal and above-diagonal elements will be stored, in column-major order. We could initialize storage for the above matrix, for instance, via c(1,5,6,12,9,2). The component ix of this class shows where in mat the various columns begin. For the above case, ix would be c(1,2,4), meaning that column 1 begins at mat[1], column 2 begins at mat[2] and column 3 begins at mat[4].

The function below creates an instance of this class. Its argument inmat is in full matrix format, i.e. including the 0s. Fill in the blanks:

```r
ut <- function(inmat) {
  nr <- nrow(inmat)
  rtrn <- list()
  class(rtrn) <- "ut"
  rtrn$mat <- vector(length=sum1toi(nr))
  tmp <- matrix(0:(nr-1),nrow=nr,ncol=1)
  rtrn$ix <- ______________________________
  for (i in 1:nr) {
    ixi <- rtrn$ix[i]
    _______________________ # fill in 1 or 2 lines (I used 1)
  }
  return(rtrn)
}
```

3. (40) Here we will do a form of parallel Quicksort, using PyMPI. Recall that in the usual form of Quicksort, we take our original array x and compare all of its elements to, say, x[0], placing those which are less than x[0] in one pile and the others in a second pile. We then recurse, though we won’t do so here. Here we form p piles, where p is the number of machines. The piles come from comparison to the first p-1 elements of x. Each machine then sorts its pile, calling ordinary sort(). Node 0 then combines all the sorted piles to get the sorted version of x. Fill in the blanks:
import mpi

def makepiles(x,npls):
    base = x[:npls]
    pls = []
    base.sort()
    lb = len(base)
    # the i-th pile will begin with the ID i, plus the divider
    for i in range(lb):
        pls.append([i,base[i]])
    pls.append([lb])
    for xi in __________:
        for j in range(lb):
            if ______________:
                pls[j].append(xi)
                break
            elif j == lb-1: pls[lb].append(xi)
    return pls

def main():
    if mpi.rank == 0:
        x = [12,5,13,61,9,6,20,1] # small test case
        pls = makepiles(x,mpi.size)
    else:
        x = []
        pls = []
    mychunk = _______________
    newchunk = [] # will become sorted version of mychunk
    for pile in mychunk:
        plnum = pile.pop(0)
        pile.sort()
        ________________ # fill in 1 line
        _______________ # fill in 1 or 2 lines
    if mpi.rank == 0:
        haveitall.sort()
        _______________ # fill in any number of lines (I used 1)
        print sortedx

1.
commsim <- function(n,k,nreps) {
    noverlap <- 0
    for (rep in 1:nreps) {
        comm1 <- choosecomm(n,k)
        comm2 <- choosecomm(n,k)
        if (length(intersect(comm1,comm2)) > 0) noverlap <- noverlap + 1
    }
    return(noverlap/nreps)
}
choosecomm <- function(n,k) {
    chosen <- vector(length=k)
    remaining <- 1:n
    for (i in 1:k) {
        # choose j at random from 1..length(remaining)
        j <- floor(runif(1) * length(remaining)) + 1
        chosen[i] <- remaining[j]
        remaining <- remaining[-j]
    }
    return(chosen)
}

2.
ut <- function(inmat) {
    nr <- nrow(inmat)
    rtrn <- list()
    class(rtrn) <- "ut"
    rtrn$mat <- vector(length=sum1toi(nr))
    # actually, easier to replace the next 2 lines by
    # rtrn$ix <- sum1toi(0:(nr-1)) + 1
    tmp <- matrix(0:(nr-1),nrow=nr,ncol=1)
    rtrn$ix <- apply(tmp,1,sum1toi) + 1
    for (i in 1:nr) {
        ixi <- rtrn$ix[i]
        rtrn$mat[ixi:(ixi+i-1)] <- inmat[1:i,i]
    }
    return(rtrn)
}
# returns 1+...+i
sum1toi <- function(i) return(i*(i+1)/2)
import mpi

# makes npls quicksort
def makepiles(x,npls):
    base = x[:npls]
    pls = []
    base.sort()
    lb = len(base)
    # the i-th pile will consist of the ID i, plus the divider
    for i in range(lb):
        pls.append([i,base[i]])
    pls.append([lb])
    for xi in x[npls:]:
        for j in range(lb):
            if xi <= base[j]:
                pls[j].append(xi)
                break
        elif j == lb-1: pls[lb].append(xi)
    return pls

def main():
    if mpi.rank == 0:
        x = [12,5,13,61,9,6,20,1] # small test case
        # divide x into piles to be disbursed to the various nodes
        pls = makepiles(x,mpi.size)
    else:
        x = []
        pls = []

    mychunk = mpi.scatter(pls)
    newchunk = [] # will become sorted version of mychunk
    for pile in mychunk:
        plnum = pile.pop(0)
        pile.sort()
        newchunk.append([plnum]+pile)

    haveitall = mpi.gather(newchunk)
    mpi.barrier()
    if mpi.rank == 0:
        haveitall.sort()
        sortedx = [z for q in haveitall for z in q[1:]]
        print sortedx # small test case; otherwise, write to disk or use