Exercises for session 8: Interfacing to C.

1. The files nnfind.c, bin_heap.c, bin_heap.h, item.h implement an algorithm based on k-d trees for finding nearest neighbours. Write an R interface to the functions in nnfind.c:

   ```c
   void within_neighbours(const double *X, int *pNx, const int *pp, int *neighbours, double *dists)
   ```

   and

   ```c
   void between_neighbours(const double *X, int *pNx, const double *Y, const int *pNy, const int *pp, int *neighbours, double *dists)
   ```

   In these functions X and Y are matrices of points in p-dimensional space, *pNx is the number of rows in X, *pNy is the number of rows in Y, *pp is the dimension of the space (number of columns in X and Y), neighbours is used to return the row number of the nearest neighbour (from 0 to (n-1)) and dists returns the distance to the nearest neighbour.

   The difference between the two functions is that within_neighbours finds the nearest neighbour in X of each point in X and between_neighbours finds the nearest neighbour in X of each point in Y. This means that neighbours and dists have length *pNx in within_neighbours and *pNy in between_neighbours.

   The row numbers returned are C row numbers from 0 to (n-1); you need to add 1 to get R row numbers.

   Test the code by drawing a scatterplot and connecting each point to its nearest neighbour (with `segments()`). A nice data example is `data(faithful)`.

2. A `box-car` filter is a simple smoother; on a scatterplot of (X_1, Y_1), (X_2, Y_2), ... (Y_n, X_n), it provides a smooth line illustrating how Y changes with X. Formally, for given radius r, at point x it is evaluated as:

   \[ Y_{\text{smooth}}(x) = \frac{\sum_{i:|X_i-x|<r} Y_i}{\sum_{i:|X_i-x|<r} 1}, \]

   in other words, it is the average of the Y's that have X's within r of x. Typically, we evaluate the box-car filter at x=X_1, X_2, ... X_n.

(continues...)
In R, one simple way to implement the box-car filter is the following:

```r
boxcar <- function(Y, X, radius, n=length(Y)){
  y.smooth <- rep(0,n)
  x <- 0
  for (i in 1:n){
    count <- 0
    x <- X[i]
    for (j in 1:n){
      if(abs(X[j]-x)<radius){
        count <- count+1
        y.smooth[i] <- y.smooth[i] + Y[j]
      }
    }
    y.smooth[i] <- y.smooth[i]/count
  }
  y.smooth
}
```

Try this code, for n=1000 data points, and then n=10,000. What takes the time? Code this approach in C, and see how much faster it becomes.

For keen people; a preliminary sort of the data enables you to implement this filter without the double loop; think of ‘sliding’ a window of radius r along the sorted X values. Implement the filter using this observation, and see what speed improvement you can achieve.