# RELATIVE SORTING ALGORITHM EFFICIENCIES EXERCISE

## Selection Sort

This exercise is based on the problem of sorting a list of numbers, which is one of the classic computing problems. Note that R has an excellent sorting function, **`sort()`,** which we will also be using.

To judge the effectiveness of a sorting algorithm, we count the number of *comparisons* that are required to sort a vector x of length n. That is, we count the number of times we evaluate logical expressions of the form **`x[i] < x[j].`** The fewer comparisons required, the more efficient the algorithm.

The simplest but least efficient sorting algorithm is selection sort. The selection sort algorithm uses two vectors, an unsorted vector and a sorted vector, where all the elements of the sorted vector are less than or equal to the elements of the unsorted vector. The algorithm proceeds as follows:

1. Given a vector **`x`**, let the initial unsorted vector **`u`** be equal to **`X`**, and the initial sorted vector **`s`** be a vector of length 0.
2. Find the smallest element of **`u`** then remove it from **`u`** and add it to the end of **`s`**.
3. If **`u`** is not empty then go back to step 2.

Write an implementation of the selection sort algorithm. To do this you may find it convenient to write a function that returns the *index* of the smallest element of a vector.

## Insertion Sort

Like selection sort, insertion sort uses an unsorted vector and a sorted vector, moving elements from the unsorted to the sorted vector one at a time. The algorithm is as follows:

1. Given a vector **`x`**, let the initial unsorted vector **`u`** be equal to **`X`**, and the initial sorted vector **`s`** be a vector of length 0.
2. Remove the last element of **`u`** and insert it into **`s`** so that **`s`** is still sorted.
3. If **`u`** is not empty then go back to step 2.

Write an implementation of the insertion sort algorithm. To do this you do not usually have to look at every element of the vector. Instead, if you start searching from the end, you just need to find the first *i* such that a >= $b_i$, then the new sorted vector is ($b_1$,…,$b_i$,a,$b_i$+1,…,$b_k$).

## Bubble Sort

Bubble sort is quite different from selection sort and insertion sort. It works by repeatedly comparing adjacent elements of the vector x = ($a_i$,…,$a_n$), as follows:

1. For *i* = 1,…,n – 1, if $a_i > a_{i+1}$ then swap $a_i$ and $a_{i+1}$.
2. If you did any swaps in step 1, then go back and do step 1 again.

Write an implementation of the bubble sort algorithm.

## Quick Sort Revisited

We looked at quick sort in the class sessions. Here is another implementation of quick sort. Use this implementation in the final part of this exercise.

```
quick.sort <- function(x) {
  # sort x using quicksort algorithm
  # if length(x) <= 1 then already sorted
  if (length(x) <= 1) return(x)
  # can now assume length(x) >= 2
  smalls <- c() # elements of x < x[1]
  bigs <- c()   # elements of x >= x[1]
  for (i in 2:length(x)) {
    if (x[i] < x[1]) {
      smalls <- c(x[i], smalls)
    } else {
      bigs <- c(x[i], bigs)
    }
  }
  # recursively use quicksort to sort smalls and bigs
  return(c(quick.sort(smalls), x[1], quick.sort(bigs)))
}

quick.sort2 <- function(x) {
  # version of quicksort using preallocated arrays
  # uses the conventions that y[0] == c() and y[0:k] == y[1:k]
(k >= 1)
  # if length(x) <= 1 then already sorted
  if (length(x) <= 1) return(x)
  # can now assume length(x) >= 2
  smalls <- rep(0, length(x)-1)
  bigs <- rep(0, length(x)-1)
  j <- 0  # smalls[1:j] are elements of x < x[1]
  k <- 0  # bigs[1:k] are elements of x >= x[1]
  for (i in 2:length(x)) {
    if (x[i] < x[1]) {
      j <- j + 1
      smalls[j] <- x[i]
    } else {
      k <- k + 1
      bigs[k] <- x[i]
    }
  }
  # recursively use quicksort to sort smalls and bigs
  return(c(quick.sort(smalls[0:j]), x[1],
quick.sort(bigs[0:k])))
}
```

## Compare Efficiencies of Sorting Algorithms

Use the **`system.time()`** function to compare the computational efficiencies of your four sorting algorithms. Include the R-native function **`sort()`** in each comparison. Perform these comparisons on a vector x with a uniform distribution that you create with the following R script. What conclusions can you draw?

```
set.seed(600617)
# comparison using a short vector
x <- runif(300)
print(system.time(bubble.sort(x)))
print(system.time(selection.sort(x)))
print(system.time(insertion.sort(x)))
print(system.time(quick.sort(x)))
print(system.time(quick.sort2(x)))
print(system.time(sort(x)))  # R-native function

# using a longer vector
x <- runif(5000)
print(system.time(selection.sort(x)))
print(system.time(insertion.sort(x)))
print(system.time(quick.sort(x)))
print(system.time(quick.sort2(x)))
print(system.time(sort(x)))  # R-native function

# and an even longer vector
x <- runif(50000)
print(system.time(quick.sort(x)))
print(system.time(quick.sort2(x)))
print(system.time(sort(x)))  # R-native function
```