Recently a query by Kevin Ummel on the R-help mailing list prompted a discussion of a problem that boils down to comparing the elements of two numeric vectors, `x` and `y`, and determining for each element in one vector the number of elements in the second vector that are less than or equal to it.

There are various ways of doing this. The original poster used

```
R> f1 <- function(x, y)
+     sapply(x, function(i) length(which(y < i)))
```

Richard Heiberger and Marc Schwartz both suggested

```
R> f2 <- function(x, y)
+     colSums(outer(y, x, '<'))
```

Gustavo Carvalho suggested the equivalent of

```
R> f3 <- Vectorize(function(x, y) sum(y < x), "x")
```

and Bill Dunlap, drawing on his encyclopedic knowledge of S-PLUS and R functions, noted that this operation was essential what is done in R's `findInterval` function which uses compiled code implementing a binary search.

```
R> f4 <- function(x, y) length(y) - findInterval(-x, rev(-sort(y)))
```

For large vectors `x` and `y`, Bill's version is much faster than any of the other suggestions which involve comparing each element of `x` to each elements of `y`. Interestingly, the second version (`f2`), which was suggested by two experienced R users, can become deadly slow on moderate sized vectors, because of the way that the `outer` function is implemented.

Even with moderate sized vectors [...?...]

```
R> benchmark(f1(x,y), f2(x,y), f3(x,y), f4(x,y),
+            columns=c("test","elapsed","relative"),
+            order="relative", replications=10L)
      test elapsed relative
4 f4(x, y)   0.040    1.000
1 f1(x, y)  16.904  422.600
3 f3(x, y)  18.309  457.725
2 f2(x, y)  56.969 1424.225
```

We will eliminate all but Bill Dunlap's method on this evidence and change the rules a bit. The question posed by Sunduz Keles regarded $p$-values from a test sample relative to a larger reference sample. She had a large sample from a reference distribution and a test sample (both real-valued in her case) and she wanted, for each element of the test sample, the proportion of the reference sample that was less than the element. It's a type of empirical $p$-value calculation.

A first C++ approach uses `std::upper_bound` for unordered $x$:

```
R> ## version using std::upper_bound for unordered x
R> h <- cxxfunction(signature(x_="numeric", y_="numeric"), '
+ {
+     Rcpp::NumericVector x(x_),
+         y = clone(Rcpp::NumericVector(y_));
+     int n = x.size();
+     Rcpp::IntegerVector ans(n);
+     const Rcpp::NumericVector::iterator bb = y.begin(), ee = y.end();
+                 // sort (a copy of) the y values for binary search
+     std::sort(bb, ee);
+
```

```
+     for (int i = 0; i < n; i++) {
+         ans[i] = std::upper_bound(bb, ee, x[i]) - bb;
+     }
+     return ans;
+ }
+ ', plugin="Rcpp")
```

The next C++ solution assumes $x$ is non-decreasing, and still uses `std::upper_bound`:

```
R> ## version using std::upper_bound when x is non-decreasing
R> j1 <- cxxfunction(signature(x_="numeric", y_="numeric"), '
+ {
+     Rcpp::NumericVector x(x_),
+         y = clone(Rcpp::NumericVector(y_));
+     int n = x.size();
+     Rcpp::IntegerVector ans(n);
+     const Rcpp::NumericVector::iterator bb = y.begin(), ee = y.end();
+     Rcpp::NumericVector::iterator mm = y.begin();
+                 // sort (a copy of) the y values for binary search
+     std::sort(bb, ee);
+
+     for (int i = 0; i < n; i++) {
+         mm = std::upper_bound(mm, ee, x[i]);
+         ans[i] = mm - bb;
+     }
+     return ans;
+ }
+ ', plugin="Rcpp")
R> j <- function(x, y) {
+     ord <- order(x)
+     ans <- integer(length(x))
+     ans[ord] <- j1(x[ord], y)
+     ans
+ }
```

The third C++ solution uses a sequential search:

```
R> ## version using sequential search
R> k1 <- cxxfunction(signature(xs_="numeric", y_="numeric", ord_="integer"), '
+ {
+     Rcpp::NumericVector xs(xs_),
+         y = clone(Rcpp::NumericVector(y_));
+     int n = xs.size();
+     Rcpp::IntegerVector ord(ord_), ans(n);
+     const Rcpp::NumericVector::iterator bb = y.begin(), ee = y.end();
+     Rcpp::NumericVector::iterator yy = y.begin();
+     double *xp = xs.begin();
+                 // sort (a copy of) the y values for binary search
+     std::sort(bb, ee);
+
+     for (int i = 0; i < n && yy < ee; i++, xp++) {
+         while (*xp >= *yy && yy < ee) yy++;
+         ans[ord[i] - 1] = yy - bb;
+     }
+     return ans;
```

```
+ }
+ ', plugin="Rcpp")
R> k <- function(x, y) {
+     ord <- order(x)
+     k1(x[ord], y, ord)
+ }
```

We evaluate all methods on random data of dimension $1e+05$ and $1e+07$. We also ensure results are identical.

Finally, the timing comparison:

```
R> res
      test replications elapsed relative
4  k(x, y)           10  12.781  1.00000
2  h(x, y)           10  12.821  1.00313
3  j(x, y)           10  12.947  1.01299
1 f4(x, y)           10  30.584  2.39293
```

The three C++ solutions are reasonably close in performances.