A Case Study of Different Methods to Solve a Problem in R

Phil Spector

October 11, 2010

1 The Task

Suppose we have two vectors, `x` and `y`. The goal is to find the indices in `y` that represent the value in `y` that is closest to each value of x. As a very simple example, consider the following two vectors:

```
> x = c(1,7,3,8,2)
> y = c(15,6,9,1,4,7,2)
```

Then the first value of x is closest to the fourth value of y; the second value is closest to the sixth, the third is closest to the fifth (ties are broken using the first value encountered in y), the fourth value is closest to the third, and the last value is closest to the seventh. Thus we would return the vector

```
[1] 4 6 5 3 7
```

2 Pure R Solutions

2.1 The `outer` function

One possible way to solve this problem is provided by the `outer` function in R: the expression `abs(outer(x,y,'-'))` will return a matrix where each row of the matrix represents the absolute value of the distance between a particular value of `x`, and all the values of `y` – the `which.min` function can then be applied to each row of the matrix returned by `outer` to get the appropriate index.

To test this on our trivial sample data, we can use the following code:

```
> apply(abs(outer(x,y,'-')),1,which.min)
[1] 4 6 5 3 7
```

The problem with this approach is that it generates all the distances at once in a matrix that will become very large as the lengths of `x` and `y` increase. We can see the problems this causes by testing the method with progressively longer vectors:

```
testouter = function(n){
   x = rnorm(n)
   y = rnorm(n*2)
   system.time(apply(abs(outer(x,y,'-')),1,which.min))[3]
}
```

Here we're arbitrarily having the second vector being twice the length of the first vector, but we'd see similar results with other types of samples. Let's look at how the function performs as we increase the size of `x` and `y`:

```
> res = sapply(seq(100,1000,by=100),testouter)
> names(res) = seq(100,1000,by=100)
> res
  100   200   300   400   500   600   700   800   900  1000
0.005 0.019 0.079 0.172 0.213 0.273 0.355 0.384 0.430 0.501
```

When we increase the size from 100 to 200 (an increase of two times), we see that it takes nearly four times as much time to calculate the indices. This indicates that we have an $O(n^2)$ algorithm, which will cause problems as the sample size increases.

2.2 sapply

In the previous example, there was really no need to calculate all the distances at once – to get our answer, we can consider the values of x one at a time. Here are timings for the same sample sizes, without keeping all the distances in memory:

```
testapply = function(n){
   x = rnorm(n)
   y = rnorm(n*2)
   system.time(sapply(x,function(z)which.min(abs(z-y))))[3]
}
> res = sapply(seq(100,1000,by=100),testapply)
> names(res) = seq(100,1000,by=100)
> res
  100   200   300   400   500   600   700   800   900  1000
0.001 0.003 0.006 0.011 0.017 0.021 0.028 0.039 0.052 0.065
```

While this is certainly an improvement, it's still roughly $O(n^2)$ – it's just faster from the beginning. This indicates that, eventually, if the sample size is large enough, we'll still have some problems.

2.3 A Note about Loops

While it's often said that loops in R are inherently inefficient, this is simply not the case. The biggest problem with loops occurs when an object changes size inside a loop. If the result of a loop is "pre-allocated" before the loop begins, loops in R really aren't a problem. Here's the appropriate way to use a loop to solve a problem like this in R:

```
> result = numeric(length(x))
> for(i in 1:length(x))result[i] = which.min(abs(x[i] - y))
> result
[1] 4 6 5 3 7
```

Let's see how this performs in practice:

```
testloop = function(n){
   x = rnorm(n)
   y = rnorm(n*2)
   system.time({result = numeric(length(x))
                for(i in 1:length(x))result[i] = which.min(abs(x[i] - y))})[3]
}
> res = sapply(seq(100,1000,by=100),testloop)
> names(res) = seq(100,1000,by=100)
> res
  100   200   300   400   500   600   700   800   900  1000
0.001 0.003 0.006 0.010 0.015 0.022 0.030 0.039 0.049 0.064
```

The results are actually a little bit better than using `sapply`. Unfortunately, many new users of R would implement this same algorithm as follows:

```
testloopnull = function(n){
    x = rnorm(n)
    y = rnorm(n*2)
    system.time({result = NULL
                 for(i in 1:length(x))result = c(result,which.min(abs(x[i] - y)))})[3]
}
> res = sapply(seq(100,1000,by=100),testloopnull)
> names(res) = seq(100,1000,by=100)
> res
  100   200   300   400   500   600   700   800   900  1000
0.001 0.004 0.006 0.011 0.018 0.023 0.033 0.043 0.048 0.068
```

The results don't look much worse, but we'll examine this issue again when we do a final comparison of the methods.

3 Developing an Efficient Algorithm

If you think about the solutions we've looked at, they calculate the distance between each point in `x` and each point in `y`. But since we are always using `y` as a reference vector, we could sort `y`, record it's permutation vector, and calculate distances to the smallest point in `y` until we found an increase in the distance. Then we could return the corresponding value in the permutation vector to get the index. Here's that idea implemented in R – note that the end conditions (when a value in `x` is greater or less than any in `y`) need to be handled separately:

```
fnr = function(x,y){
   ix = order(y)
   y = sort(y)

   dofind = function(x,y,ix){
      if(x < y[1])return(ix[1])
      if(x > y[length(y)])return(ix[length(y)])
      last = abs(x - y[1])
      for(j in 2:length(y)){
         test = abs(x - y[j])
         if(test > last){
            return(ix[j - 1])
            }
         last = test
         if(j == length(y))return(ix[j])
         }
      }

    sapply(x,dofind,y,ix)
}
```

Unfortunately, calculating things like this in R, where we are not taking advantage of vectorization inside the loop are very, very slow:

```
testralgo = function(n){
   x = rnorm(n)
```

```
   y = rnorm(n*2)
   system.time(fnr(x,y))[3]
}
  100   200   300   400   500   600   700   800   900  1000
0.001 0.004 0.006 0.013 0.019 0.027 0.037 0.050 0.061 0.072
```

3.1 Implementing the Algorithm in C

The identical algorithm that we used in the previous section can be implemented in C in virtually an identical way, provided that we have a sorting routine that can return a permutation vector. Here's one such implementation:

```
void myqsort(double *x,int *ii,int n)
{
  char done;
  int i,ip,iy,iup,lp;
  static int lv[16],iv[16];
  register double y;

  lv[0] = 0;
  iv[0] = n - 1;
  ip = 0;

  while(ip >= 0)
    {if((iv[ip] - lv[ip]) < 1)
       {ip--;
        continue; }

     lp = lv[ip] - 1;
     iup = iv[ip];
     y = x[iup];
     iy = ii[iup];

     for(;;)
       {if((iup - lp) < 2)break;
        if(x[++lp] < y)continue;
        x[iup] = x[lp];
        ii[iup] = ii[lp];

        for(;;)
           {if((iup-- - lp) < 2)break;
            if(x[iup] >= y)continue;
            x[lp] = x[iup];
            ii[lp] = ii[iup];
            break; }
        }

     x[iup] = y;
     ii[iup] = iy;

     if((iup - lv[ip]) < (iv[ip] - iup))
       {lv[ip + 1] = lv[ip];
```

```
        iv[ip + 1] = iup - 1;
        lv[ip]     = iup + 1; }
     else
       {lv[ip + 1] = iup + 1;
        iv[ip + 1] = iv[ip];
        iv[ip]     = iup - 1; }

     ip++;
    }
}
```

Here's the algorithm for finding the indices written in C. Note that one had to be added to the indices so that they'll correspond to the indexing used in R:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <R_ext/Memory.h>

void myqsort(double *x,int *ii,int n);

void findnr(double *x, double *y, int *nx, int *ny, int *ans)
{
  int *ii;
  int i,j;
  double last,xnow,test;

  ii = (int*)R_alloc(*ny,sizeof(int));
  if(ii == NULL){
    fprintf(stderr,"Couldn't allocate space for index vector");
    return;
  }

  for(i=0;i<*ny;i++)ii[i] = i;
  myqsort(y,ii,*ny);

  for(i=0;i<*nx;i++){
        xnow = x[i];
        if(xnow <= y[0]){
          ans[i] = ii[0] + 1;
          continue;
          }
        if(xnow >= y[*ny - 1]){
          ans[i] = ii[*ny - 1] + 1;
          continue;
          }
        last = fabs(x[i] - y[0]);
        for(j=1;j<*ny;j++){
           test = fabs(xnow - y[j]);
           if(test > last){
             ans[i] = ii[j - 1] + 1;
             break;
             }
```

```
            last = test;
            if(j == (*ny - 1))ans[i] = ii[*ny - 1] + 1;
            }
        }
}
```

The `R_alloc` function was used so that R would free the allocated memory once the function exits.

The shared library necessary to use the function in R could be created through the command

```
R CMD SHLIB findnr.c qsort.c
```

and the R functions to call and test it:

```
findnr = function (x, y)
{
    if(!is.loaded('findnr'))dyn.load('findnr.so')
    nx = length(x)
    ny = length(y)
    ans = numeric(nx)
    .C("findnr", as.double(x), as.double(y), as.integer(nx),
        as.integer(ny), ans = as.integer(ans))$ans
}

testalgoc = function(n){
    x = rnorm(n)
    y = rnorm(n*2)
    system.time(findnr(x,y))[3]
}

> res = sapply(seq(100,1000,by=100),testalgoc)
> names(res) = seq(100,1000,by=100)
> res
  100   200   300   400   500   600   700   800   900  1000
0.000 0.001 0.001 0.001 0.001 0.002 0.002 0.002 0.003 0.004
```

We can see that this algorithm is orders of magnitudes faster than the previous ones, and, although the timings are too small to verify it, seems to be $O(n)$.

4 Comparison of the Methods

The method using `outer` and the R implementation of the more efficient algorithm are too slow to use in the following comparison, so we'll compare the `sapply` solution, the two solutions based on loops, and the C implementation of the algorithm. The process can be automated using the following function:

```
testnr = function(n1,n2){
   x = rnorm(n1)
   y = rnorm(n2)
   rbind(system.time(one <- sapply(x,function(z)which.min(abs(z-y)))),
         system.time({two =  numeric(length(x))
             for(i in 1:length(x))two[i] = which.min(abs(x[i] - y))
             }),
      system.time({three <- NULL
             for(i in 1:length(x))three = c(three,which.min(abs(x[i] - y)))
```

```
            }),
      system.time(four <- findnr(x,y)))[,3]
}
```

Testing with values from 1000 to 10000:

```
> xseq = seq(1000,10000,by=1000)
> tries = cbind(xseq,2*xseq)
> answer = apply(tries,1,function(x)testnr(x[1],x[2]))
> answer
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9] [,10]
[1,] 0.062 0.246 0.530 0.925 1.415 2.224 3.236 4.052 4.682 5.884
[2,] 0.055 0.208 0.463 0.856 1.409 2.329 3.225 3.626 4.565 5.694
[3,] 0.058 0.217 0.514 0.902 1.440 2.467 3.348 3.856 4.929 6.373
[4,] 0.004 0.014 0.037 0.056 0.087 0.132 0.180 0.239 0.306 0.381
```

These results can be plotted using the `matplot` function, giving a graphical comparison among the methods:

```
> matplot(xseq,t(answer),lty=1,type='l',main='Comparison of Four Methods')
> legend('topleft',legend=c('sapply','loop','loop null','C algo'),col=1:6,lty=1)
```

The plot is shown in Figure 1. Notice how the timings for the loop implementation starting with `NULL`

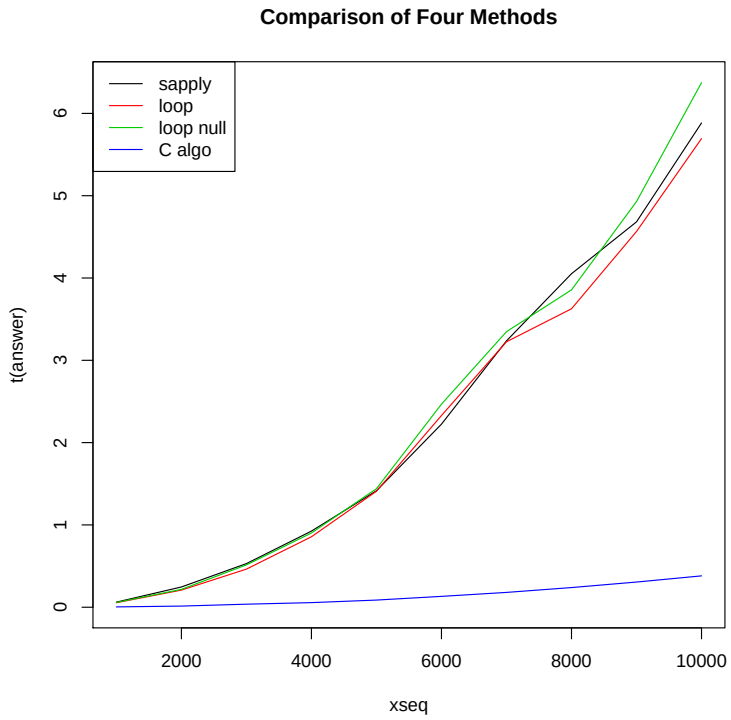

Figure 1: Comparison of the four methods

become worse and worse as `n` increases. This is due to the fact the length of the vector containing the results needs to be reallocated each time a new value is added. While this problem may not be apparent from small-scale testing, it should be avoided at all costs when writing loops in R, because it will eventually slow

things down dramatically as the sample size increases. As you might imagine, adding rows to a matrix using the `rbind` function at each iteration of a loop will cause even larger problems.