Rcpp — Solutions to Exercises

Rcpp — Exercises Part 1

- Find out why the following code gives a compile error:

```
#include<Rcpp.h>
// [[Rcpp::plugins("cpp11")]]

NumericVector x{1, 2, 3, 4, 5};
IntegerVector id{1};

double y = x[id]; // produces compile error
```

  Solution:

  `x[id]` returns a subview class of `Rcpp::Vector` which is *not* a double. It is important to note that coercion to another type is not as easily done as in `R`!

- Benchmark the functions below against each other for `x<-rnorm(1e2)`, `x<-rnorm(1e4)` and `x<-rnorm(1e6)`. Comment on the results.

```
NumericVector test_clone_return(NumericVector A) {
  NumericVector B = clone(A);
  B[1] = 0.5;
  return B;
}
```

```
NumericVector test_reference_return(NumericVector A) {
  A[1] = 0.5;
  return A;
}
```

  Solution:

```
bench::press(
  A = c(1e2, 1e4, 1e6),
  {
    A <- rnorm(A)
    bench::mark(
      test_clone_return(A),
      test_reference_return(A),
      check = F,
      relative = T
    )
  }
)
```

Obviously, cloning (that is, copying on function call) is a bad idea!

Slide 40: `Rcpp` – STL Algorithms

The second appearance of `x.begin()` refers to the beginning of the output range, i.e. the first element of `x`.

Slide 44 Rcpp — Exercises Part 2

1. Note that `f(n)` returns the n^th Fibonacci number. Here are two `Rcpp` approaches. `f_rec_cpp()` is a one-to-one implementation of the recursive `R` function `f()`. `f_cpp()` uses a loop for the computation.

   ```
   // [[Rcpp::export]]
   int f_rec_cpp(const int& n) {
     if (n < 3) return(n);
     return (f_rec_cpp(n - 1)) + f_rec_cpp(n - 2);
   }

   // [[Rcpp::export]]
   IntegerVector f_cpp(const int& n) {
     IntegerVector x(n);
     x[0] = 1;
     x[1] = 2;
     for (int i = 2; i < n; i++){
       x[i] = x[i-2] + x[i-1];
     }
     return(tail(x, 1));
   }
   ```

   Let's microbenchmark these functions:

   ```
   bench::mark(
     f = f(20),
     f_rec_cpp(20),
     f_cpp = f_cpp(20)
   )
   ```

   ```
   ## # A tibble: 3 x 6
   ##   expression         min   median `itr/sec` mem_alloc `gc/sec`
   ##   <bch:expr>    <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
   ## 1 f               2.36ms   2.41ms      411.   63.54KB     58.0
   ## 2 f_rec_cpp(20)  16.11us   16.4us    59216.    2.49KB      0
   ## 3 f_cpp         614.91ns  697.1ns   980089.    2.49KB      0
   ```

   The overhead of recursive function calling in `R` is much bigger than in `C++`. The `Rcpp` implementation `f_rec_cpp()` already gives an impressive speedup. However, using a `for` loop in `f_cpp` is even faster!

2.
   ```
   set.seed(1234)

   # let's generate a large vector to be (partially) sorted
   v <- rnorm(1e5)
   n <- 1e4

   # benchmark n^th partial sort
   bench::mark(
     nth_partial_sort(v, n),
     sort(v, partial = 1:n), check = F
   )
   ```

   ```
   ## # A tibble: 2 x 6
   ##   expression                  min   median `itr/sec` mem_alloc `gc/sec`
   ##   <bch:expr>             <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
   ## 1 nth_partial_sort(v, n) 548.78us    602us     1640.  783.79KB    25.9
   ## 2 sort(v, partial = 1:n)   6.27ms   6.38ms      156.    1.19MB     4.17
   ```

3. Here are very basic `Rcpp` versions for vector input:

```
Rcpp::cppFunction('
bool all_cpp(const LogicalVector& x) {
  int n = x.size();

  for(size_t i = 0; i < n; i++) {
    if(x[i] != true) return false;
  }

  return true;
}
')

Rcpp::cppFunction('
NumericVector range_cpp(const NumericVector& x) {
  NumericVector out = {min(x), max(x)};
  return out;
}
')

Rcpp::cppFunction('
double var_cpp(const NumericVector& x) {
   int n = x.size() - 1;
   return 1./n * sum(pow(x - mean(x), 2));
}
')
```