BIOSTAT615 Homework Assignment #1

Due : September 28th, 2017 by 8:30am (before class)

Send the following 3 files in separate attachments (do not compress or bundle) within a single email to BIOSTAT.cct3mnptvt0hcpow@u.box.com

- **[your_uniqname]_hw1a.cpp**
- **[your_uniqname]_hw1a.pdf**
- **[your_uniqname]_hw1b.py**

Note that the email address above is DIFFERENT from the address used for Homework 0, and it will be different next time too.

Read carefully the problems below to understand how to prepare each file for submission.

Multiple submissions are allowed before the due and only the last submission will be valid. Make sure that your submission was sent by checking the confirmation email.

A. Experimental analysis of sorting time complexity

1) Implement three **Rcpp** functions named "**insertion_sort**", "**bubble_sort**", "**merge_sort**" based on the skeleton code below. Do not remove any line from the skeleton and add your own code in the part instructed. Name the file as **[your_uniqname]_hw1a.cpp**, and include it for your submission.

```
/////////////////////////////////////////////////////////////////
// below is the skeleton of [your_uniqname]_hw1a.cpp file
/////////////////////////////////////////////////////////////////
#include <Rcpp.h>
#include <iostream>
#include <algorithm>
using namespace Rcpp;
using namespace std;
// [[Rcpp::export]]
NumericVector insertion_sort(NumericVector x) {
   /*
     Using insertion sort algorithm,
     make x sorted and return it.
     You may use the lecture slides or pseudo-code in the URL:
     https://www.toptal.com/developers/sorting-algorithms/insertion-sort
     to implement this function
   */
   return x;
}

// [[Rcpp::export]]
```

```
NumericVector bubble_sort(NumericVector x) {
   /*
     Using bubble sort algorithm,
     make x sorted and return it.
     You may use the  pseudo-code in the URL:
     https://www.toptal.com/developers/sorting-algorithms/bubble-sort
     to implement this function
   */
 return x;
}

void merge_sort(NumericVector x, int l, int r) {
   /*
     Using merge sort algorithm, make x sorted between indicies l ... (r-1)
     Do not attempt to return it (as oppsed to other functions)
     you may use the lecture slides or pseudo-code in the URL:
     https://www.toptal.com/developers/sorting-algorithms/merge-sort
     to implement this function
   */
}

// [[Rcpp::export]]
NumericVector merge_sort(NumericVector x) {
 // Do not modify this function.
 // This function simply calls the merge_sort function above
 // that uses divide-and-conquer algorithm
 merge_sort(x, 0, (int)x.size());
 return x;
}

// [[Rcpp::export]]
NumericVector std_sort(NumericVector x) {
 // Do not modify this function.
 // This function uses std::sort() function to perform sorting
 sort(x.begin(), x.end());
 return x;
}
/////////////////////////////////////////////////////////////////////////////
// end of [your_uniqname]_hw1a.cpp file
/////////////////////////////////////////////////////////////////////////////
```

2) First, run the following R script to produce a time complexity plot of sorting algorithms.

```
##########################################################################
## beginning of the R script to run
##########################################################################
library(Rcpp)
## make sure to change the path below to point the correct path
sourceCpp("/path/to/[your_uniqname]_hw1a.cpp")
## the list of sample sizes to test
sizes <- c(1000, 2000, 5000, 10000, 20000, 50000, 100000, 200000, 500000,
1000000, 2000000, 5000000)
## the list of function names to test
funcs <- c("insertion_sort","bubble_sort","merge_sort","std_sort","sort")
```

```
## data frame to sort the results
df <- data.frame()
for(func in funcs) { ## for each function
  prevtime <- 0      ## store the running time for the last run
  for(sz in sizes) { ## for each input size
    if ( prevtime < 10 ) { ## do not attemp if previous run time is over 10s
      x <- rnorm(sz) ## make a random values with specified length
      ## measure the CPU time, and floor it to be 0.001s for plotting
      t <- max(system.time(x <- do.call(func,list(x)))[1],0.001)
      ## make a data frame with 5 columns
      ## size : size of input data
      ## func : name of the function
      ## time : user CPU time in seconds
      ## min  : minimum value of sorted output – for verification
      ## max  : maximum value of sorted output – for verification
      ## make sure that min small (e.g. -3.xx) and max is large (e.g. 3.xx)
      df <- rbind(df, data.frame(size=sz, func=func, time=t, min=x[1],
max=x[sz]))
      prevtime <- t
    }
  }
}
## need to install ggplot2 if haven't
library(ggplot2)
## customize the name of plotted pdf file if needed.
pdf("plot.pdf",width=8,height=5)
## draw a plot between size and time, colored by function is log-log scale
print(ggplot(df,aes(x=size,y=time,colour=func))+geom_point()+geom_line()+sca
le_x_log10()+scale_y_log10())
dev.off()
print(df) ## Just to verify that min and max values reflects sorted results
#########################################################################
## end of the R script to run
#########################################################################
```

Second, write a report (in PDF format) with filename **[your_uniqname]_hw1a.pdf**, containing the answers to the following questions:

a. Include the graph generated by the R script above. Make sure to replace the file names highlighted in brown are changed appropriately to produce a valid plot. Also, check the results of last line – print(df) – to ensure that the sorted results make sense.
b. Solely from the graph, describe whether the time complexity of each of the five algorithms will be $\Theta(n^2)$ or $\Theta(n \log n)$. Justify your answer.
c. Compare the computational time between the methods with the same time complexity. What do you think the reasons for the differences are?

B. Changing Money

Suppose you are traveling in a country where currency is available only in bills and/or coins of a number of different floating values $f_1 > f_2 > \cdots > f_k > 0$.

You need to write a python program named **[your_uniqname]_hw1b.py**, which, given an floating-point vallue $v > 0$, lists all different ways that an amount of money of v can be provided in different combinations of bills and/or coins.

You may assume that all input arguments are positive floating-point values and v is the first argument followed by $f_1, f_2, \cdots, f_k$ as the remaining arguments, without having to handle incorrect inputs that do not conform to these requirements.

The first line of your output should contain the total number of possible combinations. If the number is positive, starting from the second line, you need to list all the specific combinations, where the number of bills or coins of each values are printed. Example runs of valid input arguments are as below:

```
$ python [your_uniqname]_hw1b.py 5 2 1
There are 3 possible ways to make 5.0
{1.0: 5}
{1.0: 3, 2.0: 1}
{1.0: 1, 2.0: 2}

$ python [your_uniqname]_hw1b.py 13 2
There are 0 possible ways to make 13.0

$ python [your_uniqname]_hw1b.py 2.25 1.0 0.25 0.1 0.05
There are 166 possible ways to make 2.25
{0.25: 1, 1.0: 2}
{1.0: 2, 0.1: 2, 0.05: 1}
{1.0: 2, 0.1: 1, 0.05: 3}
{1.0: 2, 0.05: 5}
{0.25: 5, 1.0: 1}
{0.25: 4, 1.0: 1, 0.1: 2, 0.05: 1}
{0.25: 4, 1.0: 1, 0.1: 1, 0.05: 3}
{0.25: 4, 1.0: 1, 0.05: 5}
{0.25: 3, 1.0: 1, 0.1: 5}
{0.25: 3, 1.0: 1, 0.1: 4, 0.05: 2}
{0.25: 3, 1.0: 1, 0.1: 3, 0.05: 4}
{0.25: 3, 1.0: 1, 0.1: 2, 0.05: 6}
{0.25: 3, 1.0: 1, 0.1: 1, 0.05: 8}
{0.25: 3, 1.0: 1, 0.05: 10}
...
```

Hint 1 : Use a divide and conquer algorithm to solve the problem.
Hint 2 : Allow some numerical precision errors when comparing two floating-point numbers.
Using 1e-10 as maximum precision errors are usually reasonable for comparing between double-precision values.