BIOSTAT615 Homework Assignment #3

Due : November 2nd, 2017 by 8:30am (before class)

Send the following 4 files in separate attachments (do not compress or bundle) within a single email to BIOSTAT.ywc0wczt236fsw65@u.box.com

- **`[your_uniqname]_hw3a.cpp`**
- **`[your_uniqname]_hw3a.pdf`**
- **`[your_uniqname]_hw3b.py`**
- **`[your_uniqname]_hw3c.cpp`**

Note that the email address above is DIFFERENT from the address used for Homework 0 to 2, 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, but only the last submission will be considered valid. Make sure that your submission was sent by checking the confirmation email. You must send the email from your [your_uniqname]@umich.edu email account.

Note that there no late submission will be allowed, unless late submission is pre-approved by documented reasons (with medical records, doctor's notes, or requests from advisor)

A. Computing binomial coefficients

Write an Rcpp function **`choose_dp(n,k)`** to calculate the binomial coefficient $\binom{n}{k}$ based on a given skeleton file **`hw3a_skeleton.cpp`**, and store the file with name **`[your_uniqname]_hw3a.cpp`**, (do not include **`'['`** or **`']'`** in the filename).

The binomial coefficient is defined as $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ and the Rcpp function **`choose_fac(n,k)`** implements the definition.

In the function **`choose_dp(n,k)`**, you need to use the following recursive rule to compute the binomial coefficient.

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$
$$\binom{n}{0} = \binom{n}{n} = 1$$

To implement this, you may define a new C++ function (but NOT another Rcpp function with **`[[Rcpp::export]]`**). You may not use import any additional libraries or packages except for those already included.

The skeleton Rcpp source code, named **`hw3a_skeleton.cpp`**, is also copied below. Note that you may not modify **`fac()`** and **`choose_fac()`** functions.

```
#include <Rcpp.h>
#include <map>
#include <algorithm>

using namespace Rcpp;
using namespace std;

//////////////////////////////////////
// Skeleton code for Homework 3A
//////////////////////////////////////

// DO NOT MODIFY: function to calculate factorial
int fac(int n) {
  if ( n < 2 ) return 1; // return 1 when n=0,1
  int ret = 1;
  for(int i=2; i <= n; ++i)
    ret *= i;  // calculate factorial
  return ret;
}

// DO NOT MODIFY: compute choose(n,k) using factorial
// [[Rcpp::export]]
int choose_fac(int n, int k) {
  return fac(n)/fac(k)/fac(n-k);
}

// MODIFY: compute choose(n,k) using dynamic programming
// [[Rcpp::export]]
int choose_dp(int n, int k) {
  ////////////////////////////////////////////////
  // TODO: FILL IN THE BLANK to return choose_dp(n,k)
  // following the dynamic programming algorithm.
  // You may define an additional function.
  ////////////////////////////////////////////////
}
```

Next, write a report (in PDF format) with filename **`[your_uniqname]_hw3a.pdf`**, including the answers to the following questions:

a. Include the results of **`choose_fac(), choose_dp(), choose()`** with the following combinations of parameters:
   (1) n = 10, k = 5
   (2) n = 15, k = 5
   (3) n = 20, k = 10

(4) n = 30, k = 15
(5) n = 60, k = 30

b. Explain why the output appears as is. Especially, if the answer is not consistent between the methods, explain which result is correct and justify why incorrect results appears for specific functions.

B. Dynamic programming implementation of changeMoney problem..

The solutions for the changeMoney problem in Homework 1B is available as **hw1b_solution.py**. We will modify the algorithm using dynamic programming. The requirement is to avoid redundant calculation from the previous implementation. The skeleton code, which is provided as **hw3b_skeleton.py**, is also pasted below.

```
import sys
import copy

################################################################
## Skeleton code of dynamic programming version of changeMoney
## Arguments:
## - total  : (integer) total amount of money to make
## - unit   : list of containing units of bills in integer
## - stored : dictionary containing a stored information to avoid
##            redundancy
################################################################
def dpChangeMoney(total, units, stored):
################################################################
## TODO: implement the function above.
################################################################

dollars = int(sys.argv[1])
bills   = [int(x) for x in sys.argv[2:len(sys.argv)]]
ways = dpChangeMoney(dollars, bills, {})
print("There are " + str(len(ways)) + " possible ways to make
"+str(dollars))
for way in ways:
    print(sorted(way.items()))
```

Here are the minimum requirements to obtain full credits:

(1) You may not import new libraries or define a new function.
(2) The output of the new algorithm should be exactly the same with the output from **hw1b_solution.py**. Note the way how the output is printed is slightly changed from the original version in Homework 1B (both for **hw1b_solution.py** and **hw3b_skeleton.py**).
(3) The new algorithm should run substantially faster than the original algorithm for complex inputs. Especially, for arguments **300 100 50 20 10 5 2 1** (for making $300 for existing dollar bills), the new algorithm should run at least twice faster than the original algorithm in the **scs** server. You may test the time efficiency by yourself by running the following command.
$ time python [program_name] 300 100 50 20 10 5 2 1 | head

Note that the input arguments are integers only, so any fractional inputs will not be considered, and there is no need to check for numerical precision errors.

Write your final program with filename **[your_uniqname]_hw3b.py**.
Hint: You probably need to use **copy.deepcopy()** *function at least once in your solution.*

C. Edit distance with alignment, with arbitrary costs.

Implement a dynamic programming algorithm that computes edit distance while providing information for alignment between the strings. A difference from the lecture slide is that the edit distance contributed by single-character mismatch and single character insertion (or deletion) are not necessarily one, and should be provided as integer arguments **`mcost`** and **`icost`**, respectively.

The return value should be a **`Rcpp::List`** containing the following four attributes

- **`distance`**: the optimal edit distance
- **`first`**: a copy of first string input argument, adding **`'-'`** where the optimal alignment is insertion (**`I`**) from the first to the second string,
- **`alignment`**: a string representing the optimal alignment between strings, with **`'.'`** when the aligned character matches, **`'*'`** when the aligned character mismatches, **`'I'`** when a character is inserted from the first to the second string, **`'D'`** when a character is deleted from the first to the second string,
- **`second`**: a copy of the second string input argument, adding '**-**' where the optimal alignment is deletion (**`D`**) from the first to the second string.

A skeleton code, which is provided with filename **`hw3c_skeleton.cpp`**, is also copied below. Modify the code and rename it to **`[your_uniqname]_hw3c.cpp`** and submit it. You may not add any new function of modify existing part of the skeleton code.

```
#include <Rcpp.h>
#include <iostream>
#include <string>

using namespace Rcpp;
using namespace std;

//////////////////////////////////////////////////////////////////////////////
//
// editDistance() function
// - Note that this is not an Rcpp function, but a function
//   only called within C++
// - Arguments:
//   > s1 : The first (reference) string to align
//   > s2 : The second (alternative) string to align
//   > cost : A matrix to contain cost to each node (negative if not stored)
//   > move : A matrix to contain the optimal move to each node (negative if
not stored)
//   > r  : row    index (0..s1.size()), indicating the position in s1
//   > c  : column index (0..s1.size()), indicating the position in s2
//   > mcost : cost of single letter mismatch
//   > icost : cost of single latter insertion/deletion
// - Return value:
//   > the optimal edit distance
// - Expected behavior:
//   > cost matrix will store the optimal cost from (0,0)...(r,c)
```

```
//   > move matrix will store the optimal previous move from (0,0)...(r,c)
//////////////////////////////////////////////////////////////////////////////
//
int editDistance(string& s1, string& s2, IntegerMatrix& cost, IntegerMatrix&
move, int r, int c, int mcost, int icost) {
  ////////////////////////////////////////////////////////
  // TODO : FILL IN THE BLANK to satisfy the specs above
  ////////////////////////////////////////////////////////
}

//////////////////////////////////////////////////////////////////////////////
// alignWords() function (Exported by Rcpp)
// - Arguments:
//   > s1 : The first (reference) string to align
//   > s2 : The second (alternative) string to align
//   > mcost : Cost of single letter mismatch
//   > icost : Cost of single latter insertion/deletion
// - Return values: A List containing the following attributes:
//   > distance  : Optimal edit distance between the string
//   > first     : A modified version of the first string to include gaps
//                 as '-' so that attributes have same length
//   > alignment : Alignment between letters
//   >             '.' indicates match,
//   >             '*' indicates mismatch,
//   >             'I' indicates insertion from first to second string
//   >             'D' indicates insertion from first to second string
//   > second    : A modified version of the first string to include gaps
//                 so that attributes have same length
//
//   For example, an example can be
//   distance  : 6
//   first     : ALGORI-THM
//   alignment : ..***.I.**
//   second    : ALTRUISTIC
//////////////////////////////////////////////////////////////////////////////
// [[Rcpp::export]]
List alignWords(string s1, string s2, int mcost, int icost) {
  ////////////////////////////////////////////////////////
  // TODO : FILL IN THE BLANK to satisfy the specs above
  ////////////////////////////////////////////////////////
}
```

An example output can be found below. Note that there are usually multiple sets of optimal alignments, so the alignment may not necessarily match to your implementation.

```
> alignWords("FOOD","MONEY",1,1)
$distance
[1] 4
$first
[1] "FO-OD"
$alignment
[1] "*.I**"
$second
[1] "MONEY"
```

```
> alignWords("FOOD","MONEY",3,1)
$distance
[1] 7
$first
[1] "-FO---OD"
$alignment
[1] "ID.IIIDD"
$second
[1] "M-ONEY--"

> alignWords("ALGORITHM","ALTRUISTIC",1,1)
$distance
[1] 6
$first
[1] "ALGORI-THM"
$alignment
[1] "..***.I.**"
$second
[1] "ALTRUISTIC"

> alignWords("ALGORITHM","ALTRUISTIC",2,1)
$distance
[1] 9
$first
[1] "ALGOR-I-THM"
$alignment
[1] "..D*.I.I.**"
$second
[1] "AL-TRUISTIC"
```