Biostatistics 615/815 - Statistical Computing
Lecture 1 : Introduction to Statistical Computing

Hyun Min Kang

September 6th, 2011

BIOSTAT615/815 - Objectives

1. Equip the ability to IMPLEMENT computational/statistical IDEAS into working SOFTWARE PROGRAMs
   - ✓ Understand the concept of algorithm
   - ✓ Understand basic data structures and algorithms
   - ✓ Practice the implementation of algorithms into programming languages
   - ✓ Develop ability to make use of external libraries

BIOSTAT615/815 - Objectives

1. Equip the ability to IMPLEMENT computational/statistical IDEAS into working SOFTWARE PROGRAMs
2. Learn COMPUTATIONAL COST management in developing statistical methods.
   - ✓ Understand the practical importance of computation cost in many statistical inference applications.
   - ✓ Develop the ability to estimate computational time and memory required for an algorithm given data size.
   - ✓ Develop the ability to improve computation efficiency of existing algorithms and to optimize the cost/accuracy trade-off.

BIOSTAT615/815 - Objectives

1. Equip the ability to IMPLEMENT computational/statistical IDEAS into working SOFTWARE PROGRAMs
2. Learn COMPUTATIONAL COST management in developing statistical methods.
3. Understand NUMERICAL and RANDOMIZED ALGORITHMS for statistical inference
   - ✓ Learn numerical optimization methods for solving analytically intractable problems computationally
   - ✓ Understand a variety of randomized algorithms for robust and efficient estimation of computationally intractable problems to obtain deterministic solution.

Why Is Statistical Computing Important?

1. Wherever there is a statistical application, that is where you'll need computation.
   - ✓ For example, typical regression or maximum likelihood estimation requires nontrivial computational procedures.
   - ✓ R or SAS may do the computation for you, but you need to understand what they are doing exactly.
2. Computational efficiency is critical for large-scale data analysis
   - ✓ In many analyses of high throughput biological data, more accurate methods cannot used in practice due to prohibitive computational cost.
   - ✓ Many statistical methods should work in principle if you have infinite time, but almost impossible to run with large-scale data due to exponential time complexity with data size.
   - ✓ Algorithmic understating of statistical methods may turn apparently impossible task into a possible one.

What Will Be Covered?

1. C++ Basics and Introductory Algorithms
   - Computational Time Complexity
   - Sorting
   - Divide and Conquer Algorithms
   - Searching
   - Key Data Structure and Standard Template Libraries
   - Dynamic Programming
   - Hidden Markov Models

What Will Be Covered?

1. C++ Basics and Introductory Algorithms
2. Numerical Methods and Randomized Algorithms
   - Random Numbers
   - Matrix Operations and Least Square Methods
   - Importance Sampling
   - Expectation-Maximization
   - Markov-Chain Monte Carlo (MCMC) Methods
   - Simulated Annealing
   - Gibbs Sampling

Textbooks

- *"Introduction to Algorithms"* (Strongly Recommended)
  - ✓ by Cormen, Leiserson, Rivest, and Stein (CLRS)
  - ✓ Third Edition, MIT Press, 2009
- *"Numerical Recipes"* (Recommended)
  - ✓ by Press, Teukolsky, Vetterling, and Flannery
  - ✓ Third Edition, Cambridge University Press, 2007
- *"C++ Primer Plus"* (Optional)
  - ✓ by Stephen Prata
  - ✓ Fifth Edition, Sams, 2004

Assignments and Grading

BIOSTAT615

- Biweekly Assignments - 50%
- Midterm Exam - 25%
- Final Exam - 25%

BIOSTAT815

- Biweekly Assignments - 33%
  - Expected to solve extra problems on top of 615 assignments
- Midterm Exam - 17%
- Final Exam - 17%
- Projects, to be completed in pairs - 33%

Target Audience for BIOSTAT615

- Target audience for BIOSTAT615 includes those with little or small programming experience, with strong motivation to learn programming statistical methods in C++ language.
- Audiences should be familiar with the concept of probability distribution, hypothesis testing, and linear regression (BIOSTAT601 is not strictly required).
- Even if you have no experience in C/C++/Java, you can follow the class, but you will need to expect to spend additional hours to accomplish the homework, especially for the first few weeks of the course.

Target Audience for BIOSTAT815

- Target Audience for BIOSTAT815 includes those with decent programming experience, with strong motivation to practice advanced methods in statistical computing throughout the lectures and the course project.
- Audiences should be familiar with programming languages, so that they can accomplish additional problem and class project.
- List of suggested projects will be announced in the next lecture.

Choice of Programming Language for Homework Assignments

- Strongly recommend to use C++.
- If you want to use other programming languages, C and Java are accepted, but you will likely to spend additional hours to port the problems code into your preferred languages.

Honor code

- Honor code is strongly enforced throughout the course. Remember that one of the main objective of the course is to equip the ability to implement statistical methods ON YOUR OWN.
- You are NOT allowed to share any piece of your homework with your colleagues electronically (e.g. via E-mail or IM)
- You may partially help debugging your colleagues' homework by sharing your trial and errors, but you may provide help only verbally in person. The help should affect only non-significant fraction of the homework. If significant fraction of your homework or course project is contributed by someone else, you must notify to your instructor so that only your contribution can be reflected into the assessment.
- If a break of honor code is identified, your entire homework (or exam) will be graded as zero, while incomplete submission of homework assignment will be considered for partial credit.

More information

Office hours

- Friday 9:00-10:30AM.
- Utilize this time slot actively for questions.

Course Web Page

- Visit
  - ✓ *http://genome.sph.umich.edu/wiki/Biostatistics_615/815*
  - ✓ or *http://goo.gl/9DoFo*

Algorithms

An Informal Definition

- An **algorithm** is a sequence of well-defined computational steps
- that takes a set of values as **input**
- and produces a set of values as **output**

Key Features of Good Algorithms

- Correctness
  - ✓ Algorithms must produce correct outputs across all legitimate inputs
- Efficiency
  - ✓ Time efficiency : Consume as small computational time as possible.
  - ✓ Space efficiency : Consume as small memory / storage as possible
- Simplicity
  - ✓ Concise to write down & Easy to interpret.

The "Old MacDonald" Song

- *http://www.youtube.com/watch?v=hDt_MhIKpLM*

A Generalization of "Old MacDonald" Song

Input

animals: cow, sheep, pig, duck, horse

noises: moo, baa, oink, quack, neigh

Output

1. Sing "Old MacDonald had a farm E I E I O"
2. Sing "And on that farm he had a $animals[1]$ E I E I O"
3. Sing "With a $noises[1]$ $noises[1]$ here, and $noises[1]$ $noises[1]$ there, here a $noises[1]$ there $noises[1]$, everywhere a $noises[1]$ $noises[1]$"
4. Sing "Old MacDonald had a farm E I E I O"
5. ······

Algorithm SingOldMacDonald

```
Data: animals[1 ··· n], noises[1 ··· n]
Result: An "Old MacDonald" Song with animals and noises
for i = 1 to n do
    Sing "Old MacDonald had a farm, E I E I O";
    Sing "And on that farm he had some animals[i], E I E I O";
    Sing "With a noises[i] noises[i] here, and a noises[i] noises[i] there";
    Sing "Here a noise[i], there a noise[i], everywhere a noise[i] noise[i]";
    for j = i − 1 downto 1 do
        Sing "noise[j] noise[j] here, noise[j] noise[j] there";
        Sing "Here a noise[j], there a noise[j], everywhere a noise[j] noise[j]";
    end
    Sing "Old MacDonald had a farm, E I E I O.";
end
```

(adapted from Jeff Erickson(UIUC)'s class notes)

Analysis of Algorithm SINGOLDMACDONALD

Correctness

- Need a formal definition of the "Old MacDonald" song for proof.
- Proof by induction
  - By showing the algorithm produces the same song with the formal definition (details are omitted)

Analysis of Algorithm SINGOLDMACDONALD

Time Complexity

- Count how many words the algorithm produces
- For each i
  - First four lines produces 41 words
  - Two lines of inner loop produces 16 words for each j
  - The last line produces 10 words
- $T(n) = \sum_{i=1}^{n} \left(51 + \sum_{j=1}^{i-1} 16\right) = 43n + 8n^2$ words are produced.
- Asymptotic complexity is quadratic : $T(n) = \Theta(n^2)$
  - When n is sufficiently large, a single order magnitude increase of n results in two orders of magnitude increase in $T(n)$.

Summary - Algorithm SINGOLDMACDONALD

- An informal toy example of an algorithm.
- Algorithm as a series of instructions to produce expected output given input data
- Generalized to handle a variety set of well-defined inputs
- Fairly high-level description of algorithm - "Sing" is the unit function of the algorithm.

Sorting

Sorting - A Classical Algorithmic Problem

The Sorting Problem

Input A sequence of n numbers. $A[1 \cdots n]$

Output A permutation (reordering) $A'[1 \cdots n]$ of input sequence such that $A'[1] \leq A'[2] \leq \cdots \leq A'[n]$

Sorting Algorithms

- Insertion Sort
- Selection Sort
- Bubble Sort
- Shell Sort
- Merge Sort
- Heapsort
- Quicksort
- Counting Sort
- Radix Sort
- Bucket Sort
- And much more..

A Visual Overview of Sorting Algorithms

http://www.sorting-algorithms.com

Today - Insertion Sort

http://www.sorting-algorithms.com/insertion-sort

Key Idea of Insertion Sort

- For k-th step, assume that elements $a[1], \cdots, a[k-1]$ are already sorted in order.
- Locate $a[k]$ between index $1, \cdots, k$ so that $a[1], \cdots, a[k]$ are in order
- Move the focus to $k+1$-th element and repeat the same step

Algorithm InsertionSort

Data: An unsorted list $A[1 \cdots n]$
Result: The list $A[1 \cdots n]$ is sorted

```
for j = 2 to n do
    key = A[j];
    i = j - 1;
    while i > 0 and A[i] > key do
        A[i + 1] = A[i];
        i = i - 1;
    end
    A[i + 1] = key;
end
```

Correctness of InsertionSort

Loop Invariant

At the start of each iteration, $A[1 \cdots j-1]$ is loop invariant iff:

- $A[1 \cdots j-1]$ consist of elements originally in $A[1 \cdots j-1]$.
- $A[1 \cdots j-1]$ is in sorted order.

A Strategy to Prove Correctness

Initialization Loop invariant is true prior to the first iteration

Maintenance If the loop invariant is true at the start of an iteration, it remains true at the start of next iteration

Termination When the loop terminates, the loop invariant gives us a useful property to show the correctness of the algorithm

Correctness Proof (Informal) of InsertionSort

Initialization

- When $j = 2$, $A[1 \cdots j-1] = A[1]$ is trivially loop invariant.

Maintenance

If $A[1 \cdots j-1]$ maintains loop invariant at iteration j, at iteration $j+1$:

- $A[j+1 \cdots n]$ is unmodified, so $A[1 \cdots j]$ consists of original elements.
- $A[1 \cdots i]$ remains sorted because it has not modified.
- $A[i+2 \cdots j]$ remains sorted because it shifted from $A[i+1 \cdots j-1]$
- $A[i] \leq A[i+1] \leq A[i+2]$, thus $A[1 \cdots j]$ is sorted and loop invariant

Termination

- When the loop terminates $(j = n+1)$, $A[1 \cdots j-1] = A[1 \cdots n]$ maintains loop invariant, thus sorted.

Time Complexity of InsertionSort

Worst Case Analysis

Algorithm	Cost
for $j = 2$ **to** n	$c_1 n$
do	
$key = A[j]$;	$c_2(n-1)$
$i = j - 1$;	$c_3(n-1)$
while $i > 0$ *and* $A[i] > key$	$c_4 \sum_{j=2}^{n} j$
do	
$A[i+1] = A[i]$;	$c_5 \sum_{j=2}^{n} (j-1)$
$i = i - 1$;	$c_6 \sum_{j=2}^{n} (j-1)$
end	
$A[i+1] = key$;	$c_7(n-1)$
end	

$$
\begin{aligned}
T(n) &= \frac{c_4 + c_5 + c_6}{2} n^2 + \frac{2(c_1 + c_2 + c_3 + c_7) + c_4 - c_5 - c_6}{2} n - (c_2 + c_3 + c_4 + c_7) \\
&= \Theta(n^2)
\end{aligned}
$$

Summary - Insertion Sort

- One of the most intuitive sorting algorithm
- Correctness can be proved by induction using loop invariant property
- Time complexity is $O(n^2)$.

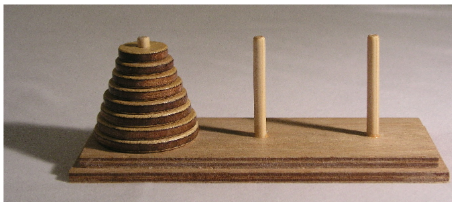
Tower of Hanoi

Problem

Input
- A (leftmost) tower with n disks, ordered by size, smallest to largest
- Two empty towers

Output Move all the disks to the rightmost tower in the original order

Condition
- One disk can be moved at a time.
- A disk cannot be moved on top of a smaller disk.



How many moves are needed?

A Working Example

http://www.youtube.com/watch?v=aGlt2G-DC8c

Think Recursively

Key Idea

- Suppose that we know how to move $n-1$ disks from one tower to another tower.
- And concentrate on how to move the largest disk.

How to move the largest disk?

- Move the other $n-1$ disks from the leftmost to the middle tower
- Move the largest disk to the rightmost tower
- Move the other $n-1$ disks from the middle to the rightmost tower

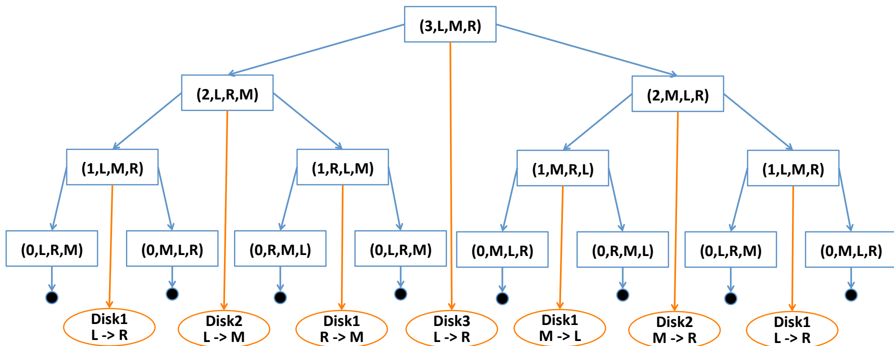
A Recursive Algorithm for the Tower of Hanoi Problem

Algorithm TowerOfHanoi

Data: n : # disks, (s, i, d) : source, intermediate, destination towers
Result: n disks are moved from s to d

```
if n == 0 then
    do nothing;
else
    TowerOfHanoi(n - 1, s, d, i);
    move disk n from s to d;
    TowerOfHanoi(n - 1, i, s, d);
end
```

How the Recursion Works



Analysis of TowerOfHanoi Algorithm

Correctness

- Proof by induction - Skipping

Time Complexity

- $T(n)$: Number of disk movements required
  - ✓ $T(0) = 0$
  - ✓ $T(n) = 2T(n-1) + 1$
- $T(n) = 2^n - 1$
- If $n = 64$ as in the legend, it would require $2^{64} - 1 = 18,446,744,073,709,551,615$ turns to finish, which is equivalent to roughly 585 billion years if one move takes one second.

Example C++ Development Environment

1. UNIX / gcc environment
    - Instructor's preference
    - UNIX environment will be commonly used in large-scale data analysis, so it would be good to be familiar with it.
    - Ways to set up UNIX environment
        - Install Linux (e.g. Ubuntu) locally to your computer
        - Download and install Xcode in Mac OS X, and use terminal to access UNIX interface.
        - Install Cygwin to a windows machine (mimics UNIX environment)
        - Connect to U-M login service via SSH using PuTTY or similar software (Refer to *http://www.itd.umich.edu/login/* for details)
    - Learning Unix-cultured editors such as vi or emacs is also recommended.

Example C++ Development Environment

1. UNIX / gcc environment
2. Windows / Microsoft Visual C++
3. Windows / Borland C++ Builder
4. Mac OS X / Xcode development environment

Getting Started with C++

Writing helloWorld.cpp

```
#include <iostream> // import input/output handling library
int main(int argc, char** argv) {
  std::cout << "Hello, World" << std::endl;
  return 0; // program exits normally
}
```

Compiling helloWorld.cpp

Install Cygwin (Windows), Xcode (MacOS), or nothing (Linux).

```
user@host:~/$ g++ -o helloWorld helloWorld.cpp
```

Running helloWorld

```
user@host:~/$ ./helloWorld
Hello, World
```

Implementing TowerOfHanoi Algorithm in C++

towerOfHanoi.cpp

```
#include <iostream>
#include <cstdlib>
// recursive function of towerOfHanoi algorithm
void towerOfHanoi(int n, int s, int i, int d) {
  if ( n > 0 ) {
    towerOfHanoi(n-1,s,d,i); // recursively move n-1 disks from s to i
    // Move n-th disk from s to d
    std::cout << "Disk " << n << " : " << s << " -> " << d << std::endl;
    towerOfHanoi(n-1,i,s,d); // recursively move n-1 disks from i to d
  }
}
// main function
int main(int argc, char** argv) {
  int nDisks = atoi(argv[1]); // convert input argument to integer
  towerOfHanoi(nDisks, 1, 2, 3); // run TowerOfHanoi(n=nDisks, s=1, i=2, d=3)
  return 0;
}
```

Running TowerOfHanoi Implementation

Running towerOfHanoi

```
user@host:~/$ ./towerOfHanoi 3
Disk 1 : 1 -> 3
Disk 2 : 1 -> 2
Disk 1 : 3 -> 2
Disk 3 : 1 -> 3
Disk 1 : 2 -> 1
Disk 2 : 2 -> 3
Disk 1 : 1 -> 3
```

Implementing InsertionSort Algorithm

insertionSort.cpp - main() function

```
#include <iostream>
#include <vector>
void printArray(std::vector<int>& A);    // declared here, defined later
void insertionSort(std::vector<int>& A); // declared here, defined later
int main(int argc, char** argv) {
  std::vector<int> v; // contains array of unsorted/sorted values
  int tok;            // temporary value to take integer input
  while ( std::cin >> tok ) // read an integer from standard input
    v.push_back(tok)        // and add to the array
  std::cout << "Before sorting:";
  printArray(v);    // print the unsorted values
  insertionSort(v); // perform insertion sort
  std::cout << "After sorting:";
  printArray(v);    // print the sorted values
  return 0;
}
```

Implementing InsertionSort Algorithm

insertionSort.cpp - printArray() function

```
// print each element of array to the standard output
void printArray(std::vector<int>& A) { // call-by-reference : will explain later
  for(int i=0; i < A.size(); ++i) {
    std::cout << " " << A[i];
  }
  std::cout << std::endl;
}
```

Implementing InsertionSort Algorithm

insertionSort.cpp - insertionSort() function

```
// perform insertion sort on A
void insertionSort(std::vector<int>& A) { // call-by-reference
  for(int j=1; j < A.size(); ++j) { // 0-based index
    int key = A[j];  // key element to relocate
    int i = j-1;     // index to be relocated
    while( (i >= 0) && (A[i] > key) ) { // find position to relocate
      A[i+1] = A[i]; // shift elements
      --i;           // update index to be relocated
    }
    A[i+1] = key;    // relocate the key element
  }
}
```

Running InsertionSort Implementation

Test with small-sized data (in Linux)

```
user@host:~/$ seq 1 20 | shuf | ./insertionSort
Before sorting:  18 9 20 3 1 8 5 19 7 16 17 12 2 15 14 10 13 6 11 4
After sorting:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Running time evaluation with large data

```
user@host:~/$ time sh -c 'seq 1 100000 | shuf | ./insertionSort > /dev/null'
real 0m24.615s
user 0m24.650s
sys 0m0.000s
user@host:~/$ time sh -c 'seq 1 100000 | shuf | /usr/bin/sort -n > /dev/null'
real 0m0.238s
user 0m0.250s
sys 0m0.020s
```

/usr/bin/sort is orders of magnitude faster than insertionSort

Summary

- Algorithms are sequences of computational steps transforming inputs into outputs
- Insertion Sort
  - ✓ An intuitive sorting algorithm
  - ✓ Loop invariant property
  - ✓ $\Theta(n^2)$ time complexity
  - ✓ Slower than default `sort` application in Linux.
- A recursive algorithm for the Tower of Hanoi problem
  - ✓ Recursion makes the algorithm simple
  - ✓ Exponential time complexity
- C++ Implementation of the above algorithms.

Homework 0

- Implement the following two programs and send the output screenshots to the instructor (hmkang at umich dot edu) by E-mail
  1. HelloWorld.cpp
  2. TowerOfHanoi.cpp
- Briefly describe your operating system and C++ development environment with your submission
- This homework will not be graded, but mandatory to submit for everyone who wants to take the class for credit
- No due date, but homework 0 must be submitted prior to submitting any other homework.

Next Lecture

Reading Materials

- CLRS Chapter 1-2 (pp. 3-42)

What to expect

- C++ Programming 101
- Fisher's exact test