Objectives

- Understanding computational aspects of statistical methods.
  - Estimate computational time and memory required
  - Understand how the method scales with data size

- Learning practical skills for efficient implementation of methods.
  - Determine appropriate data structure for implementation
  - Make use of existing libraries when useful.
  - Implement one’s own library / routine when necessary

- Developing algorithmic perspective for improving analytic methods.
  - Approximation algorithms for computationally intractable problems.
  - Computational improvement of existing methods
Why Study Statistical “Computing”?

- Statistical methods need to “compute” from data.
  - Need to understand computation for better interpretation of the results.

- Computational efficiency is critical for large-scale data analysis
  - In genomic data analysis, more accurate methods are often not used in practice due to prohibitive computational cost.
  - Many algorithms work “in principle”, but almost impossible to run with large-scale data due to exponential time complexity with data size.

- Many statistical methods require “optimization” or “randomization”
  - Logistic regression
  - Maximum-likelihood estimation
  - Bootstrapping
  - Markov-chain Monte Carlo (MCMC) methods
What Will Be Covered?

1. Algorithms 101
   - Computational Time Complexity
   - Sorting
   - Divide and Conquer Algorithms
   - Searching
   - Key Data Structure
   - Dynamic Programming
What Will Be Covered?

2. Matrices and Numerical Methods

- Matrix decomposition (LU, QR, SVD)
- Implementation of Linear Models
- Numerical optimizations
What Will Be Covered?

3. Advanced Statistical Methods

- Hidden Markov Models
- Expectation-Maximization
- Markov-Chain Monte Carlo (MCMC) Methods
Textbooks

Required Textbook

- “Introduction to Algorithms”
  - by Cormen, Leiserson, Rivest, and Stein (CLRS)

Optional Textbooks

- “Numerical Recipes”
  - by Press, Teukolsky, Vetterling, and Flannery

- “C++ Primer Plus”
  - by Stephen Prata
  - Fifth Edition, Sams, 2004
Assignments

BIOSTAT615

- Weekly Assignments - 50%
- Midterm Exam - 20%
- Final Exam - 30%

BIOSTAT815

- Weekly Assignments - 33%
- Midterm Exam - 14%
- Final Exam - 20%
- Projects, to be completed in pairs - 33%
Target Audiences

**BIOSTAT615**

- Programming experience is not required
- Those who do not have previous programming experience should expect to spend additional time studying and learning to be familiar with a programming language during the coursework.

**BIOSTAT815**

- Students should be familiar with programming languages, so that they can accomplish class project.
- List of suggested projects will be announced shortly.
Choice of Programming Language

- C++ is preferred.
- C or Java is acceptable, but may require additional work.
More information

Office hours

- Fill-in doodle poll at http://doodle.com/7z2mqvft8cdhh4bn

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.
An Informal Example

Old MacDonald Song

http://www.youtube.com/watch?v=7_mol6B9z00

Algorithm SINGOLDMACDONALD (from Jeff Erickson’s notes)

Data: animals[1 \cdots n], noises[1 \cdots 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 this 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 "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
Analysis of Algorithm \texttt{SingOldMacDonald}

**Correctness**

- Need a formal definition of the “Old MacDonald” song for proof.
- Prove by showing the algorithm produces the same song with the formal definition

**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 of $T(n) = \Theta(n^2)$. 
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
Insertion Sort

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

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
        $i = i - 1$;
    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


### 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 $A[i + 1 \cdots j - 1]$.

### 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**

```plaintext
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
```

\[
T(n) = \frac{c_4 + c_5 + c_6}{2} n^2 + \frac{2(c_1 + c_2 + c_3 + c_7)}{2} + c_4 - c_5 - c_6 n - (c_2 + c_3 + c_4 + c_7)
\]

\[
= \Theta(n^2)
\]
## 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 $d$

```plaintext
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.
Getting Started with C++

Writing helloWorld.cpp

```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
```
#include <iostream>

// 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**

```cpp
#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;
}
```

---

Hyun Min Kang
Biostatistics 615/815 - Lecture 1
Januray 6th, 2011 30 / 35
Implementing **InsertionSort** Algorithm

---

**insertionSort.cpp - printArray() function**

```cpp
// 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

```cpp
// perform insertion sort on A
void insertionSort(std::vector<int>& A) { // call-by-reference
    for(int j=1; j < A.size(); ++j) { // θ-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
    }
}
```

Hyun Min Kang
Running **INSERTIONSORT** Implementation

### Test with small-sized data (in Linux)

```bash
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

```bash
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.
For the Next Lecture

Reading Materials

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

What to expect

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