Course Objective

- Introduce skills required for executing statistical computing projects

- Applications and examples mostly in C.
  - Can be easily translated into R, etc.

- But the focus is on an algorithmic way of thinking!
Part I: Key Algorithms

- Connectivity
- Sorting
- Searching
- Hashing
- Key data structures
Part II: Statistical Methods

- Random Numbers
- Markov-Chain Monte-Carlo
  - Metropolis-Hastings
  - Gibbs Sampling
- Function Optimization
  - Naïve algorithms
  - Newton’s Methods
  - E-M algorithm
- Numerical Integration
Textbooks

- Algorithms in C
  - Sedgewick (1998)
  - 3rd edition printed in 1998

- Numerical Recipes in C
  - Press, Teukolsky, Vetterling, Flannery
  - 2nd edition printed in 2002
Assessment for 615

- Weekly Assignments
  - About 50% of the final mark

- 2 Exams
  - About 50% of the final mark
Assessment for 815

- Weekly Assignments
  - About 33% of the final mark

- 2 Exams
  - About 33% of the final mark

- Project, to be completed in pairs
  - About 33% of the final mark
Office Hours

- Please fill out doodle poll with your availability:
  - www.doodle.ch/tsuhaiaqe4ar5cer

- My office:
  - School of Public Health II, Crossroads Level 4

- My e-mail:
  - goncalo@umich.edu
Algorithms

- Methods for solving problems that are well suited to computer implementation
- Good algorithms make apparently impossible problems become simple
Algorithms are ideas ...

- Focus on approach to a problem

- Typically, the actual implementation could be take many different forms
  - Computer languages
  - Pen and paper
Example:
DNA Sequence Matches

- When the Human Genome Project started, searching through the entire genome sequence seemed impractical...

- For example,
  - Searching for ~150 sequences of about 500bp each in ~3,000,000,000 bases of sequence would take ~3 hours with the original BLAST or FASTA3 algorithms
Example: DNA Sequence Matches

- Mullikin and colleagues (2001) described an improved algorithm, using hash tables, that could do this in < 2 seconds

- Reference:
Today’s Lecture

- Introduce a “Connectivity problem” and some alternative solutions

- If you haven’t done much programming before, don’t worry too much about implementation details.

  • We’ll fill these in later lectures.
The Connectivity Problem

- **N objects**
  - Integer names 0 .. N – 1

- **M connections between pairs of objects**
  - Each connection identifies a pair \((p, q)\)

- **Possible questions:**
  - Are all objects connected?
  - Are some connections redundant?
  - What are the groups of connected objects?
Possible applications

- Is a direct connection between two computers required in a network?
  - Or can we use some existing connections instead?

- Are two individuals part of the same extended family in a genetic study?

- Are two genes in the same regulatory network?
Are the two points connected?
A possible approach ...

- Process connections one-by-one

- First, check whether a connection links two previously unconnected items
  - If not, proceed to the next connection
  - If yes, update list of connected items

- With N items, no more than N-1 updates to the list of connected items required...
  - After N-1 updates, all items connected to each other!
A simple example ...

- Connections
  - 3-4
  - 4-9
  - 8-0
  - 2-3
  - 5-6
  - 2-9
  - 4-8
  - 0-2
A simple example ...

- Connections
  - 3-4 √
  - 4-9 √
  - 8-0 √
  - 2-3 √
  - 5-6 √
  - 2-9 Redundant: 2-3 ; 3-4 ; 4-9
  - 4-8 √
  - 0-2 Redundant: 0-8; 8-4; 4-3; 3-2
Specific Tasks

- As we proceed through list of connections, conduct two tasks:
  - Decide if each connection is new.
  - Incorporate information about new connections.
The Fundamental Operations

- The *Find* operation
  - Identify the set containing a particular item or items.

- The *Union* operation
  - Replace the sets containing two groups of objects by their union
The First Step

- Developing a solution that works
  - Easy to verify correctness
  - May not be most efficient
  - Should be simple

- Useful as check of “better” solutions…
Arrays of Integers

- Simple data structure
  - Analogous to a vector

- The notation $a[i]$ refers to the $i^{th}$ integer in the array
  - When programming, we typically pre-specify the total number of entries in a array
Quick Find Algorithm

- **Data**
  - Array of N integers
  - Objects \( p \) and \( q \) connected iif \( a[p] == a[q] \)

- **Setup**
  - Initialize \( a[i] = i \), for \( 0 \leq i < N \)

- **For each pair**
  - If \( a[p] == a[q] \) objects are connected (FIND)
  - Move all entries in set \( a[p] \) to set \( a[q] \) (UNION)
A Simple C Implementation

```c
#define N 1000

int main()
{
    int i, p, q, set, a[N];
    int unique_connections = 0;

    for (i = 0; i < N; i++)
        a[i] = i;

    while (read_connection(p, q))
    {
        if (a[p] == a[q]) continue;

        set = a[p];
        for (i = 0; i < N; i++)
            if (a[i] == set)
                a[i] = a[q];

        print_connection(p, q);
        unique_connections++;
    }
    return 0;
}
```
**A Simple C Implementation**

```c
bool read_connection(int &a, int &b)
{
    // The scanf function is a standard C function for processing input data. It processes input fields (each marked with a % sign) according to a type qualifier. For example, %d fields are stored in integer variables.

    bool success = scanf(" %d %d ", &a, &b) == 2;

    return success;
}

void print_connection(int a, int b)
{
    // The printf function is a standard C function for processing output. It processes fields (each marked with a % sign) according to a type qualifier. For example, %d fields are replaced with the contents of corresponding integer variables.

    printf("The connection from %d to %d is non-redundant.\n", a, b);
}
```
Pictorial Representation

- Array as connections are added:
  - 3-4
  - 4-9
  - 8-0
  - 2-3
  - 2-9 * Redundant *
How efficient is Quick Find?

- If there $N$ objects and $M$ connections*, the Quick Find algorithm requires on the order of $MN$ operations.
- Not feasible for very large numbers of objects…

* In this case only non-redundant connections actually count.
Quick-Union Algorithm I

- Complementary to Quick Find
- More complex data organization
  - Each object points to “parent” object in the same set
Quick-Union Algorithm II

- For each pair
  - Follow pointers until we reach object that points to itself
  - If \(a[p]\) and \(a[q]\) eventually lead to the same object, we are in the same set (FIND)
  - Otherwise, link the object to which \(a[p]\) leads to the object which \(a[q]\) leads (UNION)
// Loop through connections
while (read_connection(p, q))
{
    // Check that input is within bounds
    if (p < 0 || p >= N || q < 0 || q >= N) continue;

    // FIND operation
    i = a[p];
    while (a[i] != i)
        i = a[i];

    j = a[q];
    while (a[j] != j)
        j = a[j];

    if (i == j) continue;

    // UNION operation
    a[i] = j;

    print_connection(p, q);
    unique_connections++;
}
Pictorial Representation

Array as connections are added:

- 3-4
- 4-9
- 8-0
- 2-3
- 2-9 * Redundant *
How efficient is Quick Union?

- Quick Union is typically faster than Quick Find.

- However, the data can conspire to make things difficult:
  - If objects are paired 1-2; 2-3; 3-4; 4-5; ... we’ll build long chains which slow down FIND operations.

- In the worst case, we might even need more than MN operations.
Weighted Quick Union

- A smarter version of Quick Union, that avoids long chains

- Keep track of the number of elements in each set (using a separate array)

- Link smaller set to larger set
  - Union increases length of chains in smaller set by 1
C Implementation

// Initialize weights
for (i = 0; i < N; i++)
    weight[i] = 1;

// Loop through connections
while (read_connection(p, q))
{
    // Check that input is within bounds
    if (p < 0 || p >= N || q < 0 || q >= N) continue;

    // FIND operation
    for (i = a[p]; a[i] != i; i = a[i] ) ;
    for (j = a[q]; a[j] != j; j = a[j] ) ;
    if (i == j) continue;

    // UNION operation
    if (weight[i] < weight[j])
        { a[i] = j; weight[j] += weight[i]; }
    else
        { a[j] = i; weight[i] += weight[j]; }

    print_connection(p, q);
    unique_connections++;
}
Pictorial Representation

- Array as connections are added:
  - 3-4
  - 4-9
  - 8-0
  - 2-3
  - 2-9 * Redundant *
Efficiency of Weighted Quick Union

- Guarantees that pointer chains are no more than $\log_2 N$ elements long
- Overall, requires about $M \log_2 N$ operations
- Suitable for very large data sets with millions of objects and connections
Pictorial Comparison
Quick Union      Quick Find      Weighted
## Empirical Timings in Seconds

<table>
<thead>
<tr>
<th>Nodes (Connections)</th>
<th>Quick Find</th>
<th>Quick Union</th>
<th>Weighted Quick Union</th>
</tr>
</thead>
<tbody>
<tr>
<td>50,000 (50,000)</td>
<td>6</td>
<td>1</td>
<td>&lt;1</td>
</tr>
<tr>
<td>100,000 (100,000)</td>
<td>12</td>
<td>4</td>
<td>&lt;1</td>
</tr>
<tr>
<td>200,000 (200,000)</td>
<td>25</td>
<td>15</td>
<td>&lt;1</td>
</tr>
</tbody>
</table>
Summary

- Considered 3 alternative solutions to the “connectivity problem”
  - Are any connections in a set redundant?
  - Are all objects in a set connected?

- Compared some of the computational cost for the different methods
Reading Material

- Read Chapter 1 of Sedgewick
- www.sph.umich.edu/csg/abecasis/class/