Notes on Problem Set 1

- Results were very positive!
  - (But homework was time-consuming!)

- Familiar with Union Find algorithms

- Language of choice
  - 50% tried C
  - 50% tried R
Question 1

How many random pairs of connections are required to connect 1,000 objects?

- Answer: ~3,740

Useful notes:
- Count all connections
- Use simple termination condition
Question 2

- What are path lengths in the saturated tree?
  - ~1.8 nodes on average
  - ~5 nodes for maximum path

- Random data is better than worst case
  - $\log_2 N$ nodes
Is it better to use tree height or weight in ordering union operations?

- Tree weight is better. If we point the root of a tree with $X$ nodes to the root of a tree with $Y$ nodes, the length of $X$ paths increases by 1.
  - Smallest increase corresponds to $X < Y$.
- However, using height ensures that longest path is shorter.
Other notes...

- Indent code
  - Easier to read
  - Easier to debug and spot mistakes

- Useful functions for editing code in R
  - `Debug()` for stepping through lines of code
  - `Edit()` opens a text-editor for a function
Last Lecture...

- Introduction to Programming in C
  - Data and function types
  - Control structures

- Standard C Function Library

- Arithmetic Precision
Today...

- Introduce recursive functions
- The Stack
- Problematic recursive functions
Recursion

- A function that is part of its own definition

\[ \begin{align*}
N \cdot \text{Factorial}(N - 1) & \quad \text{if } N > 0 \\
1 & \quad \text{if } N = 0
\end{align*} \]

- A program that calls itself
Key Applications of Recursion

- Dynamic Programming
  - Related to Markov processes in Statistics
- Divide-and-Conquer Algorithms
- Tree Processing
Recursive Function in R

Factorial <- function(N)
{
    if (N == 0)
        return(1)
    else
        return(N * Factorial(N - 1))
}
Recursive Function in C

```c
int factorial (int N)
{
    if (N == 0)
        return 1;
    else
        return N * factorial(N - 1);
}
```
Key Features of Recursions

- Initial set of known values
- Recursive definition for other values
  - Computation of large $N$ depends on smaller $N$
- Can generally be expressed with a loop
An Exception, where N increases:
A Strange Recursive Function in C

```c
int puzzle (int N)
{
    if (N == 1)
        return 1;

    if (N % 2 == 0)
        return puzzle(N / 2);

    return puzzle(3 * N + 1);
}
```
Evaluating puzzle(3)

puzzle(3)
  puzzle(10)
    puzzle(5)
      puzzle(16)
        puzzle(8)
          puzzle(4)
            puzzle(2)
              puzzle(1)
More Typical: Euclid’s Algorithm

Algorithm for finding greatest common divisor of two integers $a$ and $b$

- If $a$ divides $b$
  - $\text{GCD}(a,b)$ is $a$

- Otherwise, find the largest integer $t$ such that
  - $at + r = b$
  - $\text{GCD}(a,b) = \text{GCD}(r,a)$
Euclid’s Algorithm in R

GCD <- function(a, b)
{
  if (a == 0)
    return(b)

  return(GCD(b %% a, a))
}

Euclid’s Algorithm in C

```c
int gcd (int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b % a, a);
}
```
Evaluating GCD(4458, 2099)

\[
\begin{align*}
gcd(2099, 4458) \\
gcd(350, 2099) \\
gcd(349, 350) \\
gcd(1, 349) \\
gcd(0, 1)
\end{align*}
\]
Divide-And-Conquer Algorithms

- Common class of recursive functions
- Function
  - Processes input
  - Divides input in half
  - Calls itself recursively for at least one half
  - Order of processing and recursion may vary
Binary Search

```c
int search(int a[], int value, int start, int stop)
{
    while (stop >= start)
    {
        // Find midpoint
        int mid = (start + stop) / 2;

        // Compare midpoint to value
        if (value == a[mid]) return mid;

        // Reduce input in half!!!
        if (value < a[mid])
            stop = mid - 1;
        else
            start = mid + 1;
    }

    // Search failed
    return -1;
}
```
Recursive Binary Search

```c
int search(int a[], int value, int start, int stop)
{
    // Search failed
    if (start > stop)
        return -1;

    // Find midpoint
    int mid = (start + stop) / 2;

    // Compare midpoint to value
    if (value == a[mid]) return mid;

    // Reduce input in half!!!
    if (value < a[mid])
        return search(a, start, mid - 1);
    else
        return search(a, mid + 1, stop);
}
```
Recursive Maximum

```c
int Maximum(int a[], int start, int stop)
{
    int left, right;

    // Maximum of one element
    if (start == stop)
        return a[start];

    left = Maximum(a, start, (start + stop) / 2);
    right = Maximum(a, (start + stop) / 2 + 1, stop);

    // Reduce input in half!!!
    if (left > right)
        return left;
    else
        return right;
}
```
The Stack

- Specialized area of memory
  - Grows with each function call
  - Released when function returns

- Tracks
  - Function arguments
  - Previous program state
  - Local variables

- Size of stack limits depth of recursion
A Typical Stack

- An array with 9 elements
  \[ a[] = \{2, 3, 5, 7, 11, 17, 19, 23, 29\} \]

- Function call
  \[ \text{search}(a, 15, 0, 8) \]

- Arguments and local variables for each recursive call stored in stack ...
A Typical Stack II

<table>
<thead>
<tr>
<th>Stack</th>
<th>mid=4</th>
<th>stop=8</th>
<th>start=0</th>
<th>value=15</th>
<th>a[]</th>
<th>SEARCH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>mid=6</td>
<td>stop=8</td>
<td>start=5</td>
<td>value=15</td>
<td>a[]</td>
<td>SEARCH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>mid=4</td>
<td>stop=8</td>
<td>start=0</td>
<td>value=15</td>
<td>a[]</td>
<td>SEARCH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>mid=6</td>
<td>stop=8</td>
<td>start=5</td>
<td>value=15</td>
<td>a[]</td>
<td>SEARCH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>mid=4</td>
<td>stop=8</td>
<td>start=0</td>
<td>value=15</td>
<td>a[]</td>
<td>SEARCH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Well behaved recursions

- So far …
  - Factorial
  - Greatest Common Divisor
  - Binary Search
  - Maximum

- Situations where recursions are effective
  - Does this always hold?
A trickier example

Consider the Fibonacci numbers...

\[
Fibonacci(N) = \begin{cases} 
0 & \text{if } N = 0 \\
1 & \text{if } N = 1 \\
Fibonacci(N-1) + Fibonacci(N-2) & \text{otherwise}
\end{cases}
\]
Fibonacci Numbers

```c
int Fibonacci(int i)
{
    // Simple cases first
    if (i == 0)
        return 0;
    if (i == 1)
        return 1;

    return Fibonacci(i - 1) + Fibonacci(i - 2);
}
```
Terribly Slow!

Calculating Fibonacci Numbers Recursively

Time (seconds) vs. Fibonacci Number and Time (seconds)
What is going on? ...
Faster Alternatives

- Certain quantities are recalculated
  - Far too many times!

- Need to avoid recalculation…
  - Ideally, calculate each unique quantity once.
Dynamic Programming

- A technique for avoiding recomputation
- Can make exponential running times …
- … become linear!
Bottom-Up Dynamic Programming

- Evaluate function starting with smallest possible argument value
  - Stepping through possible values, gradually increase argument value

- Store all computed values in an array

- As larger arguments evaluated, precomputed values for smaller arguments can be retrieved
Fibonacci Numbers

```c
int Fibonacci(int i)
{
    int a[LARGE_NUMBER], j;

    a[0] = 0;
    a[1] = 1;

    for (j = 2; j <= i; j++)
        a[j] = a[j - 1] + a[j - 2];

    return a[i];
}
```
Top-Down Dynamic Programming

- Save each computed value as final action of recursive function
- Check if pre-computed value exists as the first action
```c
int Fibonacci(int i) {
    // Simple cases first
    if (saveF[i] > 0)
        return saveF[i];

    if (i <= 1)
        return 1;

    // Recursion
    saveF[i] = Fibonacci(i - 1) + Fibonacci(i - 2);
    return saveF[i];
}
```
Much less recursion now...
Limitations of Dynamic Programming

- Requires integer arguments
  - Need to index results in an array

- Small number of possible argument values
  - Need enough memory to store array
Summary

- Recursive functions
- The stack
- Dynamic programming
  - We’ll see more of this on Thursday!
Reading

- Sedgewick, Chapters 5.1 – 5.3