Notes on Problem Set 1

- Results were very positive!
  - But homework was time-consuming…

- Familiar with
  - Union Find algorithms
  - Compiling and Executing C Programs
Question 1

- How many random pairs of connections are required to connect 1,000 objects?
  - Answer: ~3,740

- Useful notes:
  - Number of non-redundant links to controls loop
  - Repeat simulation to get a better estimates
Question 2

- Path lengths in the saturated tree...
  - ~1.8 nodes on average
  - ~5 nodes for maximum path

- Random data is far from worst case
  - Worst case would be paths of $\log_2 N$ nodes

- Path lengths can be calculated using weights
Question 3

- Tree height or weight for optimal quick union operations?
  - Using height ensures that longest path is shorter.
  
  - Pointing the root of a tree with $X$ nodes to the root of a tree with $Y$ nodes, increases the average length of all paths by $X/N$.
  
  - Smallest average length and faster Find operations correspond to choosing $X < Y$.
  
  - Easiest to check if you use the same sequence of random numbers for both problems.
Last Lecture

- Principles for analysis of algorithms
  - Empirical Analysis
  - Theoretical Analysis

- Common relationships between inputs and running time

- Described two simple search algorithms
Recursive refers to ...

- A function that is part of its own definition

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

- 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

- Simple solution for a few cases
- Recursive definition for other values
  - Computation of large N depends on smaller N
- Can be naturally expressed in a function that calls itself
  - Loops are sometimes an alternative
A Typical Recursion: 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)

gcd(2099, 4458)
  gcd(350, 2099)
    gcd(349, 350)
      gcd(1, 349)
        gcd(0, 1)
Divide-And-Conquer Algorithms

- Common class of recursive functions

- Common feature
  - Process input
  - Divide input in smaller portions
  - Recursive call(s) process at least one portion

- Recursion may sometimes occur before input is processed
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;
}
```
An inefficient recursion

- Consider the Fibonacci numbers...

\[
Fibonacci(N) = \begin{cases} 
0 & \text{if } N = 0 \\
1 & \text{if } N = 1 \\
Fibonacci(N - 1) + Fibonacci(N - 2) & \text{otherwise}
\end{cases}
\]
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
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
int Fibonacci(int i)
{
    int fib[LARGE_NUMBER], j;

    fib[0] = 0;
    fib[1] = 1;

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

    return fib[i];
}
Fibonacci With Dynamic Memory

```c
int Fibonacci(int i)
{
    int * fib, j, result;

    if (i < 2) return i;

    fib = malloc(sizeof(int) * (i + 1));

    fib[0] = 0; fib[1] = 1;
    for (j = 2; j <= i; j++)
        fib[j] = fib[j - 1] + fib[j - 2];

    result = fib[i];
    free(fib);

    return result;
}
```
Fibonacci Numbers in R

Fibonacci <- function(i)
{
  if (i < 2)
    return(i)

  // Arrays in R are zero based, so ensure i >= 1
  i <- i + 1
  fib <- rep(0, i)

  fib[1] <- 0;
  fib[2] <- 1;

  for (j in seq(3,i))

  return (fib[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 i;

    // Recursion
    saveF[i] = Fibonacci(i - 1) + Fibonacci(i - 2);
    return saveF[i];
}
```
Within R functions, all assignments change only local variable by default.

Must use <<- operator to change global variable.
Fibonacci Numbers

Fibonacci <- function(i)
{
    # Simple cases first
    if (i <= 1)
        return (i)

    if (saveF[i] > 0)
        return (saveF[i])

    # Recursion
    saveF[i] <<- Fibonacci(i - 1) + Fibonacci(i - 2)
    return (saveF[i])
}
Much less recursion now...
Dynamic Programming
Top-down vs. Bottom-up

- In bottom-up programming, programmer has to do the thinking by selecting values to calculate and order of calculation.

- In top-down programming, recursive structure of original code is preserved, but unnecessary recalculation is avoided.
Examples of Useful Settings for Dynamic Programming

- Calculating Binomial Coefficients
- Evaluating Poisson-Binomial Distribution
Binomial Coefficients

- The number of subsets with $k$ elements from a set of size $N$

\[
\binom{N}{k} = \binom{N-1}{k} + \binom{N-1}{k-1}
\]

\[
\binom{N}{0} = \binom{N}{N} = 1
\]
Implementation in R

Choose <- function(N, k)
{
  M <- matrix(nrow = N, ncol = N + 1)

  for (i in 1:N)
  {
    M[i,1] <- M[i, i + 1] <- 1

    if (i > 1)
      for (j in 2:i)
        M[i,j] <- M[i - 1, j - 1] + M[i - 1, j];
  }

  return(M[N,k + 1])
}
Implementation in C

```c
int Choose(int N, int k)
{
    int i, j, M[MAX_N][MAX_N];

    for (i = 1; i <= N; i++)
    {
        M[i][0] = M[i][i] = 1;

        for (j = 1; j < i; j++)
            M[i][j] = M[i - 1][j - 1] + M[i - 1][j];
    }

    return M[N][k];
}
```
Poisson-Binomial Distribution

- $X_1, X_2, \ldots, X_n$ are Bernoulli random variables

- Probability of success is $p_k$ for $X_k$

- $\sum_k X_k$ has Poisson-Binomial Distribution
Recursive Formulation

\[ P_1(0) = 1 - p_1 \]
\[ P_1(1) = p_1 \]

\[ P_j(0) = (1 - p_j)P_{j-1}(0) \]
\[ P_j(j) = p_jP_{j-1}(j-1) \]
\[ P_j(i) = p_jP_{j-1}(i-1) + (1 - p_j)P_{j-1}(j-1) \]
Summary

- Recursive functions
- The stack
- Dynamic programming
  - Bottom-up Dynamic Programming
  - Top-down Dynamic Programming
Reading

- Sedgewick, Chapters 5.1 – 5.3

Notice:
- No class on Monday, October 4
- Public Health Symposium on “Global Health – The Challenge of Inequality” At Rackham Auditorium