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 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 C

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

\[ \text{gcd}(2099, 4458) \]

\[ \text{gcd}(350, 2099) \]

\[ \text{gcd}(349, 350) \]

\[ \text{gcd}(1, 349) \]

\[ \text{gcd}(0, 1) \]
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, value, start, mid - 1);
    else
        return search(a, value, 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];

    // Process half of the data ...
    left = Maximum(a, start, (start + stop) / 2);
    right = Maximum(a, (start + stop) / 2 + 1, stop);

    // Combine results across halves ...
    if (left > right)
        return left;
    else
        return right;
}
```
An inefficient recursion

- Consider the Fibonacci numbers...

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

Fibonacci Number

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

Time
What is going on? ...
Faster Alternatives

- Certain quantities are recalculated
  - Far too many times!

- Need to avoid recalculation…
  - Ideally, calculate each unique quantity once.
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 in C

```c
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];
}
```
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;
}
Top-Down Dynamic Programming

- Save each computed value as final action of recursive function
- Check if pre-computed value exists as the first action
Fibonacci Numbers

// Note: saveF should be a global array initialized all zeros

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];
}
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 \( N \)

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

\[
\binom{N}{0} = \binom{N}{N} = 1
\]
Bottom-Up 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];
}
```
#define MAX_N 30

int choices[MAX_N][MAX_N];

void InitChoose()
{
    int i, j;

    for (i = 0; i < MAX_N; i++)
        for (j = 0; j < MAX_N; j++)
            choices[i][j] = 0;
}
Top-Down Implementation (II)

```c
int Choose(int N, int k)
{
    // Check if this is an easy case
    if (N == k || k == 0)
        return 1;

    // Or a previously examined case
    if (choices[N][k] > 0)
        return choices[N][k];

    // If neither of the above helps, use recursion
    choices[N][k] = Choose(N - 1, k - 1) + Choose(N - 1, k);

    return choices[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

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

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

- Recursive functions
  - Arise very often in statistics

- Dynamic programming
  - Bottom-up Dynamic Programming
  - Top-down Dynamic Programming

- Dynamic program is an essential tool for statistical programming
Good Programming Practices

- Today’s examples used global variables
  - Variables declared outside a function
  - Accessible throughout the program

- In general, these should be used sparingly

- Two alternatives are:
  - Using `static` variables that are setup on the first call
  - Using C++ to group a function and its data
Function with Built-In Initialization

```c
int Choose(int N, int k)
{
    static int valid = 0, choices[MAX_N][MAX_N], i, j;

    // Check if we need to initialize data
    if (valid == 0)
    {
        for (i = 0; i < MAX_N; i++)
            for (j = 0; j < MAX_N; j++)
                choices[i][j] = 0;
        valid = 1;
    }

    // Check if this is an easy case
    if (N == k || k == 0) return 1;

    // Or a previously examined case
    if (choices[N][k] > 0) return choices[N][k];

    // If neither of the above helps, use recursion
    choices[N][k] = Choose(N - 1, k - 1) + Choose(N - 1, k);
    return choices[N][k];
}
```
class Choose
{
    public:
        // This function is called to initialize data
        // Whenever a variable of type Choose is created
        Choose();

        // This function is called to calculate N_choose_k
        int Evaluate(int N, int k);

    private:
        int choices[MAX_N][MAX_N];
};
Choose::Choose()
{
    for (int i = 0; i < MAX_N; i++)
        for (int j = 0; j < MAX_N; j++)
            choices[i][j] = 0;
}

int Choose::Evaluate()
{
    // Check if this is an easy case
    if (N == k || k == 0) return 1;

    // Or a previously examined case
    if (choices[N][k] > 0) return choices[N][k];

    // If neither of the above helps, use recursion
    choices[N][k] = Choose(N - 1, k - 1) + Choose(N - 1, k);

    return choices[N][k];
}
#include "Choose.h"

int main()
{
    Choose choices;

    // Evaluate 20_choose_10
    choices.Evaluate(20, 10);

    // Evaluate 30_choose_10
    choices.Evaluate(30, 10);

    return 0;
}
Reading

- Sedgewick, Chapters 5.1 – 5.3