Last Lecture

- Recursive functions
- The stack
- Dynamic programming
  - Bottom-up
  - Top-down
Today ...

- Dynamic programming
  - Examples
- Accessing global variables in R
- Memory allocation in C
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 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];
}
```
Implementing Function in R

- Pitfalls
  - Arrays start with element 1

- Conveniences
  - Arrays size does not have to be fixed
Fibonacci Numbers

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

  i <- i + 1
  a <- rep(0, i)

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

  for (j in seq(3,i))
    a[j] <- a[j - 1] + a[j - 2]

  return(a[i])
}
Creating and Sizing Arrays in R

- `length(v)`
  - Number of elements in array
- `c(…)`
  - Concatenates a set of variables or vectors
- `rep(x, n)`
  - Repeats a value for specified number of times
- `seq(start, stop)`
  - Generates a sequence of numbers
In C:

Arbitrary Arrays are Possible...

- Program must explicitly request memory
- Program should explicitly free memory
- Memory management provided by `<stdlib.h>`

- Pointers
  - Store a memory address
  - `int *`, `double *`, `char *`
Memory Regions for C Programs

- **Static Data**
  - Variables with one instance per program

- **The Stack**
  - Local variables, with one instance per function call

- **The Heap**
  - Memory allocated at runtime using malloc()
Pointers in C

- Declared with data type followed by *

- Can be created by...
  - Calling malloc()
  - Retrieving address of existing variable (\&var)

- Accessed using ...
  - pointer to retrieve memory address
  - *pointer to retrieve first element
  - pointer[i] to retrieve i\textsuperscript{th} element
Basic Memory Management

- `void * malloc(size_t bytes)`
  - Allocates a block of memory
  - Required amount specified in bytes
  - Pointer can be converted to appropriate type

- `void free(void * pointer)`
  - Releases memory

- `sizeof(type)`
  - Returns size of data type in bytes
int Fibonacci(int i)
{
    int * a, j, result;

    if (i < 2) return i;

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

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

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

    return result;
}
Things to Remember

- Array indexing
  - 0 .. N – 1
  - 1 .. N

- Memory allocation and pointers
  - For C users!
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

```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];
}
```
Implementing Function in R

- Within R functions, all assignments change only local variable by default
- Must use `<<-` operator to change global variable
The `<<-` operator in R

- Unlike `<-` does not create a local variable.
- Searches for variable in enclosing function or global environment.
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])
}
More on Recursive Functions

- The Binomial Distribution
- The Poisson-Binomial Distribution
- Example of an unstable recursion
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 R - Notes

- Results are stored in matrix()

- Indices start at 1
  - Intermediate results stored in M[N][k+1] to avoid element zero

- Sequences in loops can go up or down
  - Writing 2:n could have unintended consequences
  - If statement checks n before executing loop
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];
}
```
Implementation in C - Notes

- Intermediate results in 2D array

- However...
  - Each row depends only on previous row
  - Each column depends only on two columns

- Instead of storing all results in matrix...
- ... keep a vector with results for row N - 1
Implementation in C

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

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

        for (j = i; j > 0; j--)
        {
        }
    }

    return M[k];
}
```
Further refinement is possible
  • E.g. Memory allocation with malloc()

Calculations can be further reduced
  • Top-down programming is more effective.
  • (Homework question!)
Poisson-Binomial Distribution

- $X_1, X_2, \ldots, X_n$ are Bernoulli random variables
- Probability of success is $p_k$ for $X_k$
- $S=\sum_k X_k$ has Poisson-Binomial Distribution
Some Possibilities

- If $p_k = p$ then $\sum_k X_k$ follows Binomial distribution with $n$ trials and probability of success $p$

- When $\sum_k p_k$ is large, $\sum_k X_k$ can be approximated by a Poisson distribution

- In other cases, we may need to evaluate distribution exactly…
  - $p_n(i) = Pr(S_n = i)$
Some Possibilities

- If $p_k = p$ then $\sum_k X_k$ follows Binomial distribution with $n$ trials and probability of success $p$

- When $\sum_k p_k$ is large, $\sum_k X_k$ can be approximated by a Poisson distribution

- In other cases, we may need to evaluate distribution exactly…
  - $p_n(i) = \Pr(S_n = i)$
Recursive Formulation

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

\[ P_j(0) = (1 - p_j)P_{j-1}(0) \]
\[ P_j(j) = p_jP_{j-1}(j-1) \]
\[ P_j(i) = p_jP_{j-1}(i-1) + (1 - p_j)P_{j-1}(j-1) \]
An unstable recursion ...

- Some floating point calculations are numerically unstable...

- A well known example involves the “Golden Ratio”...

\[ \phi = \frac{\sqrt{5} - 1}{2} = 0.61803398 \]

\[ \phi^n = \phi^{n-2} - \phi^{n-1} \]
Calculating Powers of $\phi$

- Two possibilities

\[
\phi^n = \phi^{n-1} \phi
\]

\[
\phi^n = \phi^{n-2} - \phi^{n-1}
\]
Results Using Product Formula
Results Using Difference Formula
Relative Error on Log Scale

\[
\log(\frac{\text{abs}(\text{product} - \text{difference})}{\text{product}})
\]

Exponent vs. Logarithm of Relative Error
Today ...

- Flushed out recursive programs

- These details are important in getting your code to run...
  - Array indices
  - Memory allocation
  - Local or global variables
  - Checking accuracy of calculation