CSCI 103
More Recursion & Depth First Search

Mark Redekopp
Recursive Flood Fill

• Recall the recursive algorithm for flood fill?
  – Base case: black pixel, out-of-bounds
  – Recursive case: Mark current pixel black and then recurse on your neighbors

```c
void flood_fill(int r, int c)
{
    if(r < 0 || r > 255 )
        return;
    else if ( c < 0 || c > 255){
        return;
    }
    else if(image[r][c] == 0){
        return;
    }
    else {
        // set to black
        image[r][c] = 0;
        flood_fill(r-1,c);  // north
        flood_fill(r,c-1);  // west
        flood_fill(r+1,c);  // south
        flood_fill(r,c+1);  // east
    }
}
```
Recursive Ordering

- Give the recursive ordering of all calls for recursive flood fill assuming N, W, S, E exploration order starting at 4,4
  - From what square will you first explore to the west?
  - From what square will you first explore south?
  - From what square will you first explore east?
  - What is the maximum number of recursive calls that will be alive at any point in time?
Recursive Ordering

- Give the recursive ordering of all calls for recursive flood fill assuming N, W, S, E exploration order starting at 4,4
  - From what square will you first explore to the west?
  - From what square will you first explore south?
  - From what square will you first explore east?
  - What is the maximum number of recursive calls that will be alive at any point in time?
  - Notice recursive flood fill goes deep before it goes broad

<table>
<thead>
<tr>
<th></th>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th>0,3</th>
<th>0,4</th>
<th>0,5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analyze These!

• What does this function print?

• What does this function return for g(3122013)
Sorting

• If we have an unordered list, sequential search becomes our only choice
• If we will perform a lot of searches it may be beneficial to sort the list, then use binary search
• Many sorting algorithms of differing complexity (i.e. faster or slower)
• Bubble Sort (simple though not terribly efficient)
  – On each pass through thru the list, pick up the maximum element and place it at the end of the list. Then repeat using a list of size n-1 (i.e. w/o the newly placed maximum value)
Bubble Sort Algorithm

\[ n \leftarrow \text{length(List)}; \]
\begin{align*}
\text{for} & (i = n-2; i >= 1; i--) \\
\text{for} & (j = 1; j <= i; j++) \\
\text{if} & (\text{List}[j] > \text{List}[j+1]) \text{ then} \\
\text{swap} & \text{List}[j] \text{ and List}[j+1]
\end{align*}

Pass 1

\[
\begin{array}{cccccc}
7 & 3 & 8 & 6 & 5 & 1 \\
\text{j} & \quad \text{i} \\
3 & 7 & 8 & 6 & 5 & 1 \\
\text{swap} \\
3 & 7 & 8 & 6 & 5 & 1 \\
\text{no swap} \\
3 & 7 & 6 & 8 & 5 & 1 \\
\text{swap} \\
3 & 7 & 6 & 5 & 8 & 1 \\
\text{swap} \\
3 & 7 & 6 & 5 & 1 & 8
\end{array}
\]

Pass 2

\[
\begin{array}{cccccc}
3 & 7 & 6 & 5 & 1 & 8 \\
\text{j} & \quad \text{i} \\
3 & 7 & 6 & 5 & 1 & 8 \\
\text{no swap} \\
3 & 6 & 7 & 5 & 1 & 8 \\
\text{swap} \\
3 & 6 & 5 & 7 & 1 & 8 \\
\text{swap} \\
3 & 6 & 5 & 1 & 7 & 8 \\
\text{swap}
\end{array}
\]

Pass n-1

\[
\begin{array}{cccccc}
1 & 3 & 5 & 6 & 7 & 8 \\
\text{i} \\
1 & 3 & 5 & 6 & 7 & 8 \\
\text{swap} \\
\end{array}
\]
Recursive Sort (MergeSort)

• Break sorting problem into smaller sorting problems and merge the results at the end

• Mergesort(0..n-1)
  – If list is size 1, return
  – Else
    • Mergesort(0..n/2)
    • Mergesort(n/2+1 .. n-1)
    • Combine each sorted list of n/2 elements into a sorted n-element list
Recursive Sort (MergeSort)

- Run-time analysis
  - # of recursion levels = \( \log_2(n) \)
  - Total operations to merge each level = \( n \) operations total to merge two lists over all recursive calls
- MergeSort = \( O(n \times \log_2(n)) \)
DEPTH FIRST SEARCH
DFS Algorithm

- BFS finishes all closer vertices before moving to further vertices
- DFS finishes all further vertices before closer vertices
- DFS Approach
  - Mark as started [Gray]
  - For each visited neighbor, visit it and perform DFS on all of their unstarted [White] neighbors
  - Only then, mark as finished [Black]
- DFS is recursive!!
- Coloring/marking system avoids issues with cycles (loops) in the graph

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
**Depth First-Search**

**DFS-All(G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6. DFS-Visit (G, u, finish_list)

**DFS-Visit (G, u)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3. if v.color = WHITE then
4. DFS-Visit (G, v)
5. u.color = BLACK
6. finish_list.append(u)

Finish_list is just so you can see when we're done with a vertex.
Depth First-Search

DFS-All(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5  if u.color == WHITE then
6    DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3  if v.color = WHITE then
4    DFS-Visit (G, v)
5  u.color = BLACK
6 finish_list.append(u)

DFS-Visit(G,a):
Depth First-Search

DFS-All(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

DFS-All(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

DFS-All(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6  DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4  DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)

DFS-Visit(G,h):
DFS-Visit(G,b):
DFS-Visit(G,c):
DFS-Visit(G,a):
Depth First-Search

DFS-All(G)
1 for each vertex u
2  u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5  if u.color == WHITE then
6    DFS-Visit(G, u, finish_list)

DFS-Visit (G, u)
1  u.color = GRAY
2  for each vertex v in Adj(u) do
3    if v.color = WHITE then
4      DFS-Visit (G, v)
5  u.color = BLACK
6  finish_list.append(u)

DFS-Visit(G,g):
DFS-Visit(G,h):
DFS-Visit(G,b):
DFS-Visit(G,c):
DFS-Visit(G,a):
Depth First-Search

**DFS-All(G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5.   if u.color == WHITE then
6.     DFS-Visit (G, u, finish_list)

**DFS-Visit (G, u)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3.   if v.color = WHITE then
4.     DFS-Visit (G, v)
5. u.color = BLACK
6. finish_list.append(u)

**DFS-Visit(G,a):**
**DFS-Visit(G,b):**
**DFS-Visit(G,c):**
**DFS-Visit(G,d):**
**DFS-Visit(G,g):**
**DFS-Visit(G,h):**
Depth First-Search

DFS-All(G)
1  for each vertex u
2    u.color = WHITE
3  finish_list = empty_list
4  for each vertex u do
5    if u.color == WHITE then
6      DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1  u.color = GRAY
2  for each vertex v in Adj(u) do
3    if v.color = WHITE then
4      DFS-Visit (G, v)
5  u.color = BLACK
6  finish_list.append(u)
Depth First-Search

DFS-All(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

DFS-All(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5   if u.color == WHITE then
6     DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3   if v.color = WHITE then
4     DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

DFS-All(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

DFS-All(G)
1  for each vertex u
2    u.color = WHITE
3  finish_list = empty_list
4  for each vertex u do
5    if u.color == WHITE then
6      DFS-Visit(G, u, finish_list)

DFS-Visit(G, u)
1  u.color = GRAY
2  for each vertex v in Adj(u) do
3    if v.color = WHITE then
4      DFS-Visit(G, v)
5  u.color = BLACK
6  finish_list.append(u)

d
finish_list:

DFS-Visit(G,e):
DFS-Visit(G,f):
DFS-Visit(G,g):
DFS-Visit(G,h):
DFS-Visit(G,b):
DFS-Visit(G,c):
DFS-Visit(G,a):
Depth First-Search

DFS-All(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

DFS-All(G)
1 for each vertex u
2 \ u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5   if u.color == WHITE then
6     DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3   if v.color = WHITE then
4     DFS-Visit (G, v)
5   u.color = BLACK
6 finish_list.append(u)
Depth First-Search

DFS-All(G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 finish_list.append(u)
Depth First-Search

DFS-All(G)
1  for each vertex u
2    u.color = WHITE
3  finish_list = empty_list
4  for each vertex u do
5      if u.color == WHITE then
6        DFS-Visit (G, u, finish_list)

DFS-Visit (G, u)
1    u.color = GRAY
2    for each vertex v in Adj(u) do
3      if v.color = WHITE then
4         DFS-Visit (G, v)
5    u.color = BLACK
6    finish_list.append(u)
BFS vs. DFS Algorithm

• BFS and DFS are more similar than you think
  – Do we use a FIFO/Queue (BFS) or LIFO/Stack (DFS) to store vertices as we find them

BFS-Visit (G, start_node)
1 for each vertex u
2 u.color = WHITE
3 u.pred = nil
4 bfsq = new deque
5 bfsq.push_back(start_node)
6 while bfsq not empty
7 u = bfsq.pop_front()
8 if u.color == WHITE
9 u.color = GRAY
10 foreach vertex v in Adj(u) do
11 bfsq.push_back(v)

DFS-Visit (G, start_node)
1 for each vertex u
2 u.color = WHITE
3 u.pred = nil
4 finish_list = new deque
5 finish_list.push_back(start_node)
6 while finish_list not empty
7 u = finish_list.pop_back()
8 if u.color == WHITE
9 u.color = GRAY
10 foreach vertex v in Adj(u) do
11 finish_list.push_back(v)
Memory Benefits of DFS

• Do a BFS and then DFS from the root
  – How many vertices are you keeping track of at any point in time (either in your BFS queue or DFS stack/recursive calls)
  – BFS would yield an entire level
    • Notice at level $h$ you would have $2^{h-1}$ nodes
  – DFS only tracks the depth which is just $h$
Challenge

• Copy your Maze search PA and convert it to use recursive DFS approach
• How do you turn the search into a similar, small problem that can be called over and over (i.e. recursively)
  – Search from a single square
• Base cases:
  – Finish, Wall, already visited, (maybe out of bounds)
• Recursive case:
  – Continue search from your neighbors (much like flood fill)
• Do we still need a BFSQ?
  – No, recursive call stack keeps track of our current path
• Do we still need a predecessor array?
  – Yes, to avoid revisiting locations we've already been
  – But we don't need to know our predecessor to retrace our steps (recursive call stack has the path). We just need to know if we've been there...so maybe change the name from predecessor to 'visited' or 'marked'
Activity

• Take your Maze search PA and convert it to use DFS by...
  – Using a deque or vector for your bfsq (maybe change its name to dfsq)
  – Always remove from the back and put in the back [DFS] rather than removing from the front like in BFS
  – Test using your normal maze1.in

• Now download a new maze
  – wget http://ee.usc.edu/~redekopp/cs103/maze_dfs.in

• Before running do a manual run-through of what your code will do on the maze
  – Will it find the shortest path?
  – Will it find a path?

• Verify by running your new DFS maze search on the maze_dfs.in
Recursive Helper Functions

- Sometimes we want to provide a user with a simple interface (arguments, etc.)
- But to implement it recursively we need additional arguments to our function
- In that case, we often let the top-level, simple function call a recursive "helper" function that provides the additional arguments needed to do the work

http://cs103.usc.edu/websheets/index.php#sqrt
- Find the square root of, x, without using sqrt function...
- Pick a number, square it and see if it is equal to x
- Use a binary search to narrow down the value you pick to square
Another Example

• If you are given the value, n, and an array with n characters could you generate all the combinations of n-bit binary?

• Do so recursively!

Exercise: bin_combo_str
Recursion and DFS

• Recursion forms a kind of Depth-First Search

```cpp
void binCombos(string prefix, int len)
{
    if (prefix.length() == len)
        cout << prefix << endl;
    else {
        // recurse
        binCombos(data + "0", len);
        // recurse
        binCombos(data + "1", len);
    }
}
```
Recursion and DFS (w/ C-Strings)

- Recursion forms a kind of Depth-First Search

```c
void binCombos(char* data, int curr, int len)
{
    if(curr == len )
        data[curr] = '\0';
    else {
        // set to 0
        data[curr] = '0';
        // recurse
        binCombos(data, curr+1, len);
        // set to 1
        data[curr] = '1';
        // recurse
        binCombos(data, curr+1, len);
    }
}
```
Exercises

- Zero_sum
- Prime_products
Another Recursion Example

• What would this code print for
  – X=2, y=3
  – X=10, y=2
  – X=4, y=2

```cpp
#include <iostream>
#include <string>
using namespace std;

void mystery(int r, string pre, int n) {
    if (prefix.length() == n){
        cout << pre << endl;
    }
    else {
        for (int i=0; i < r; i++){
            char c = static_cast<char>('0'+i);
            mystery(r, prefix + c, n);
        }
    }
}

int main() {
    int base, numDigits;
    cin >> x >> y;

    string pre;
    mystery(x, pre, y);
    return 0;
}
```
Recursion and DFS (w/ C-Strings)

- Answer: All combinations of base x with y digits

```cpp
#include <iostream>
#include <string>
using namespace std;

void basen_combos(int r, string pre, int n) {
    if (prefix.length() == n) {
        cout << pre << endl;
    }
    else {
        for (int i = 0; i < r; i++) {
            char c = static_cast<char>('0' + i);
            basen_combos(r, prefix + c, n);
        }
    }
}

int main() {
    int base, numDigits;
    cin >> x >> y;
    string pre;
    basen_combos(x, pre, y);
    return 0;
}
```
Knapsack Problem

- Knapsack problem
  - You have a set of objects with a given weight and value. Suppose you have a knapsack that can hold $N$ pounds, which subset of objects can you pack that maximizes the value.
  - Example:
    - Knapsack can hold 35 pounds
    - Object A: 7 pounds, $12$ ea.
    - Object C: 4 pounds, $7$ ea.
    - Object B: 10 pounds, $18$ ea.
    - Object D: 2.4 pounds, $4$ ea.
Another Exercise

• Generate all string combinations of length n from a given list (vector) of characters

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

void all_combos(vector<char>& letters, int n) {
}

int main() {
    vector<char> letters;
    letters.push_back('U');
    letters.push_back('S');
    letters.push_back('C');
    all_combos(letters, 2);
    all_combos(letters, 4);
    return 0;
}
```