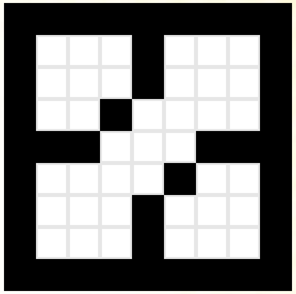
MORE RECURSION: FLOOD FILL & EXPONENTIATION

COP 3502

Recursive Flood Fill Algorithm

- A Flood Fill is a name given to the following basic idea:
  - In a space (typically 2-D, or 3-D) with an initial starting square, fill in all the adjacent squares with some value or item.
    - Until some boundary is hit.
    - For example, the paint bucket in MS Paint is an example of flood fill.



Example of a Recursive Flood Fill with 4 directions

Recursive Flood Fill Algorithm

- Imagine you want to fill in a "lake" with the ~ character.
  - We'd like to write a function that takes in one spot in the lake (the coordinates to that spot in the grid)
  - In the example, you can see we don't want to just replace all "_" with "~", because we just want to fill the contiguous area.

```
**********
**__***_**
**___*__**
**___*__**
**______**
**______**
**______**
**********
__________
__________
```

```
**********
**~~***~**
**~~~*~~**
**~~~*~~**
**~~~~~~**
**~~~~~~**
**~~~~~~**
**********
__________
__________
```

Recursive Flood Fill Algorithm

- Depending on how the floodfill should occur
  - Do we just fill in each square above, below, left, and right
  - OR do we ALSO fill in the diagonals
- The basic idea behind a recursive function, is shown in pseudocode:

```
Void FloodFill(char grid[][SIZE], int x, int y) {
      grid[x][y] = FILL_CHARACTER;

      for (each adjacent location i,j to x,y) {
            if (i,j is inbounds and not filled)
                  FloodFill(grid, i, j);
      }
}
```

Recursive Flood Fill Algorithm

- When we actually write the code,
  - We may either choose a loop to go through the adjacent locations, or simply spell them out.
  - If there are 8 locations (using the diagonal) a loop is better.
  - If there are 4 or fewer (North, South, East, West)
    - It might make more sense to write each recursive call separately.

```
Void FloodFill(char grid[][SIZE], int x, int y) {
      grid[x][y] = FILL_CHARACTER;

      for (each adjacent location i,j to x,y) {
            if (i,j is inbounds and not filled)
                  FloodFill(grid, i, j);
      }
}
```

General Structure of Recursive Functions

- Here are 2 general constructs of recursive functions

```
if (termination condition) {
    DO FINAL ACTION
}
else {
    Take 1 step closer to
    terminating condition

    Call function RECURSIVELY
    on smaller sub-problem
}
```

Typically, functions that return values use this construct.

```
if (!termination condition) {
    Take 1 step closer to
    terminating condition

    Call function RECURSIVELY
    on smaller sub-problem
}
```

While void recursive function use the this construct.

Note: These are not the ONLY layouts of recursive programs, just common ones.

Recursive Flood Fill Algorithm

- Implementation shown in class…

FAST EXPONENTIATION

COP 3502

Fast Exponentiation

- On the first lecture on recursion we discussed the Power function:
  - But this is slow for very large exponents.

```
// Pre-conditions:  exponent is >= to 0
// Post-conditions: returns base^exponent

int Power(int base, int exponent) {

       if (exponent == 0)
           return 1;
       else
           return (base*Power(base, exponent - 1);
}
```

Fast Exponentiation

- An example of an application that uses very large exponents is data encryption
  - One method for encryption of data (such as credit card numbers) involves modular exponentiation, with very large exponents.
    - Using the original recursive Power, it would take thousands of years just to do a single calculation.
    - Luckily, with one very simple observation, the algorithm can take a second or two with these large numbers.

Fast Exponentiation

- The key idea is that IF the exponent is even, we can exploit the following formula:
  - **$b^e = (b^{e/2})x(b^{e/2})$**
  - For example, $2^8 = 2^4*2^4$
    - Now, if we know 2^4 we can calculate 2^8 with a single multiplication.
    - $2^4 = 2^2*2^2$
    - And $2^2 =2*2$
  - Now we can return:
    - 2*2 = 4, 4*4 = 16, 16*16 = 256
    - This required only 3 multiplications, instead of 7

Fast Exponentiation

- The key idea is that IF the exponent is even, we can exploit the following formula:
  - **$b^e = (b^{e/2}) x (b^{e/2})$**
  - So, In order to find, b^n we find $b^{n/2}$
    - Half of the original amount
  - And then to find $b^{n/2}$, we find $b^{n/4}$
    - Again, Half of $b^{n/2}$
  - So if we are reducing the number of multiplications we have to make in half each time, what might the run time be?
    - Log n multiplications
    - Which is much better than the original n multiplications.
  - But this only works if n is even…

Fast Exponentiation

- The key idea is that IF the exponent is even, we can exploit the following formula:
  - $\mathbf{b^e = (b^{e/2})x(b^{e/2})}$
  - Since n is an integer, we have to rely on integer division which rounds down to the closest integer.
  - What if n is odd?
    - $\mathbf{b^n = b^{n/2} * b^{n/2} * b}$
    - So $\mathbf{2^9 = 2^4 * 2^4 * 2}$
  - Which gives us the following formula to base our recursive algorithm on:
    - $$b^n = \begin{cases} b^{n/2} * b^{n/2} & \text{if n is even} \\ b^{n/2} * b^{n/2} * b & \text{if n is odd} \end{cases}$$

Fast Exponentiation

- Here is the code, notice it uses the same base case as the previous Power function:

```
int PowerNew(int base, int exp) {
        if (exp == 0)
                return 1;
        else if (exp == 1)
                return base;
        else if (exp%2 == 0)
                return PowerNew(base*base, exp/2);
        else
                return base*PowerNew(base, exp-1);
}
```

Fast Exponentiation

- Here is the code for Fast Exponentiation using Mod:

```
int modPow(int base, int exp, int n) {
	base = base%n;

	if (exp == 0)
		return 1;
	else if (exp == 1)
		return base;
	else if (exp%2 == 0)
		return modPow(base*base%n, exp/2, n);
	else
		return base*modePow(base, exp-1, n)%n;
}
```

- Even using mod, the stack is overflowed quickly, so this solution needs to be translated to an iterative solution.

Practice Problem

- Print a String in reverse order:
- For example, if we want to print "HELLO" backwards,
  - we first print: "O", then we print "HELL" backwards… this is where the recursion comes in!
- See if you can come up with a solution for this

Practice Problem

- Write a recursive function that:
  - Takes in 2 non-negative integers
  - Returns the product
    - Does NOT use multiplication to get the answer
  - So if the parameters are 6 and 4
    - We get 24
    - Not using multiplication, we would have to do 6+6+6+6